# VisualApplets

User Documentation

# Imprint

Basler AG
Konrad-Zuse-Ring 28
68163 Mannheim, Germany
Tel.: +49 (0) 621 789507 0
Fax.: +49 (0) 621 789507 10

© Copyright 2024 Basler AG. All rights reserved.

Document Version: 1.9
Document Language: en (US)

Last Change: September 2024

# Table of Contents

Part I. User Manual ..... 1
1. Introduction ..... 2
1.1. VisualApplets ..... 2
1.2. How to Use This Documentation ..... 4
1.3. System Requirements ..... 4
2. Getting Started ..... 5
2.1. Writing Your First Applet ..... 5
2.2. Running Your Applet on Hardware ..... 11
2.3. Further Reading ..... 13
3. Basic Functionality ..... 14
3.1. Basic Principles ..... 14
3.2. Workflow ..... 15
3.3. Main Program Window ..... 17
3.4. Entering a Design ..... 28
3.5. Data Flow ..... 35
3.6. Rules of Links ..... 40
3.7. Diagram Parametrization ..... 53
3.8. Allocation of Device Resources ..... 70
3.9. Design Rules Check ..... 74
3.10. Simulation ..... 76
3.11. FPGA Resource Estimation ..... 108
3.12. Build ..... 112
3.13. Framegrabber SDK ..... 120
4. Extended Functionality ..... 122
4.1. Hierarchical Boxes ..... 122
4.2. User Libraries ..... 128
4.3. Custom Operator Libraries ..... 147
4.4. Multiple Processes ..... 211
4.5. Target Hardware Porting ..... 213
4.6. PixelPlant Designs ..... 214
4.7. System Settings ..... 215
4.8. Design Settings ..... 225
4.9. Build Settings ..... 227
4.10. Tcl Scripting ..... 236
4.11. Script Collection (Tcl) ..... 237
4.12. Tcl Export ..... 238
4.13. Print / Screenshot ..... 244
4.14. Migration from Older Versions ..... 245
5. Embedded VisualApplets (eVA) ..... 246
5.1. Introduction ..... 246
5.2. Common Interfaces for all Platforms ..... 253
5.3. Defining the IP Core Properties ..... 258
5.4. Embedding and Simulating the IP core ..... 302
5.5. Runtime Software Interface ..... 317
5.6. Licensing Model ..... 330
5.7. Application Notes ..... 331
6. Miscellaneous ..... 338
6.1. Command Line Options ..... 338
6.2. Keyboard Shortcuts ..... 339
6.3. Error Reporting ..... 340
Part II. Tutorial and Examples ..... 341
7. Introduction ..... 342
8. Hardware Applet: From Idea to Application ..... 343
8.1. Workflow Description ..... 343
8.2. Designing an Applet in VisualApplets ..... 343
8.3. Building the Applet in VisualApplets ..... 352
8.4. Running the Applet on Hardware ..... 353
9. Basic Design Theory ..... 358
9.1. Applet Parameterization ..... 358
9.2. Multiple DMA Channel Designs ..... 365

9.3. Synchronization of Asynchronous Image Pipelines ..... 377
10. Basic Acquisition Designs for Varying Camera Types and Hardware Platforms ..... 385
10.1. Basic Acquisition Examples for Camera Link Cameras for microEnable IV VD4-CL/-PoCL Frame Grabber ..... 387
10.2. Basic Acquisition Examples for GigE Vision Cameras for microEnable IV Frame Grabber ..... 393
10.3. Basic Acquisition Examples for Camera Link Cameras for marathon, LightBridge and ironman Frame Grabbers ..... 395
10.4. Basic Acquisition Examples for CoaXPress Cameras for marathon and ironman Frame Grabbers ..... 402
10.5. Basic Acquisition Examples for Cameras for CoaXPress 12 imaFlex Frame Grabber ..... 408
11. imaFlex CXP-12 Quad and Penta Implementation Examples ..... 412
11.1. Functional Example for the *FrameBufferMultRoiDyn* Operator on the imaFlex CXP-12 Penta Platform ..... 412
11.2. Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Penta Platform ..... 412
11.3. Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Quad Platform ..... 413
11.4. Functional Example for Loading Test Images Using *ImageInjector* ..... 413
11.5. Functional Example for Multi Tap Camera Interface with Tap Geometry Sorting. 414
11.6. Functional Example for the *JPEG_Encoder_Color_iF* User Library Element on the imaFlex CXP-12 Quad Platform ..... 414
11.7. Functional Example for the *JPEG_Encoder_Color_iF_Penta* User Library Element on the imaFlex CXP-12 Penta Platform ..... 414
11.8. Example for the *DMAFromPC* Operator on the imaFlex CXP-12 Quad Platform.. 415
12. Processing Examples ..... 416
12.1. Advanced ..... 416
12.2. Binarization ..... 429
12.3. Blob Analysis ..... 430
12.4. Color ..... 431
12.5. Co-Processor ..... 462
12.6. Debugging and Test ..... 463
12.7. Difference Images ..... 477
12.8. Filter ..... 478
12.9. Geometry ..... 483
12.10. High Dynamic Range and Image Composition ..... 519
12.11. Lookup Tables ..... 535
12.12. Loop ..... 536
12.13. Object Features ..... 542
12.14. Shading Correction ..... 556
12.15. Trigger ..... 558
13. Operator Examples ..... 577
13.1. Functional Example for Specific Operators of Library Accumulator and Library Logic ..... 577
13.2. Functional Example for Specific Operators of Library Synchronization: Dynamic Append and Cut ..... 577
13.3. Functional Example for Specific Operators of Library Memory and Library Signal ..... 578
13.4. Functional Example for Specific Operators of Library Memory and Library Signal ..... 578
13.5. Functional Example for Specific Operators of Library Signal ..... 579
13.6. Functional Example for Specific Operators of Library Synchronization, Base and Filter ..... 580
13.7. Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions ..... 580
13.8. Functional Example for Specific Operators of Library Color, Base and Memory... 581
13.9. Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters ..... 581
14. Parameter Library Examples ..... 583
14.1. Parameter Redirection ..... 583
14.2. Parameter Translation ..... 583

14.3. User Library Parameter ..... 584
14.4. Parameter Selection ..... 584
14.5. Link Parameter Translation ..... 585
15. Using Applets During Runtime ..... 586
15.1. Filling LUT with Content With the Basler Framegrabber API ..... 586
Part III. Operator Reference ..... 588
16. Introduction ..... 589
17. Library Overview ..... 590
18. Library Accumulator ..... 592
18.1. ColMax ..... 594
18.2. ColMin ..... 596
18.3. ColSum ..... 598
18.4. Count ..... 600
18.5. FrameMax ..... 603
18.6. FrameMin ..... 605
18.7. FrameSum ..... 607
18.8. Histogram ..... 609
18.9. ModuloCount ..... 611
18.10. Register ..... 615
18.11. RowMax ..... 617
18.12. RowMin ..... 619
18.13. RowSum ..... 621
19. Library Arithmetics ..... 623
19.1. ABS ..... 625
19.2. ADD ..... 627
19.3. ARCCOS ..... 629
19.4. ARCCOT ..... 632
19.5. ARCSIN ..... 635
19.6. ARCTAN ..... 638
19.7. ClipHigh ..... 641
19.8. ClipLow ..... 643
19.9. COS ..... 645
19.10. COT ..... 648
19.11. DIV ..... 651
19.12. MULT ..... 653
19.13. RND ..... 655
19.14. SCALE ..... 657
19.15. ShiftLeft ..... 660
19.16. ShiftRight ..... 662
19.17. SIN ..... 665
19.18. SQRT ..... 668
19.19. SUB ..... 669
19.20. TAN ..... 671
20. Library Base ..... 674
20.1. BRANCH ..... 677
20.2. CastBitWidth ..... 678
20.3. CastColorSpace ..... 681
20.4. CastKernel ..... 682
20.5. CastParallel ..... 686
20.6. CastType ..... 688
20.7. CONST ..... 690
20.8. ConvertPixelFormat ..... 692
20.9. Coordinate_X ..... 695
20.10. Coordinate_Y ..... 697
20.11. Dummy ..... 699
20.12. DynamicROI ..... 700
20.13. EventToHost ..... 703
20.14. EventDataToHost ..... 709
20.15. ExpandToKernel ..... 712
20.16. ExpandToParallel ..... 713
20.17. GetStatus ..... 715
20.18. HierarchicalBox ..... 716

20.19. ImageNumber ..... 718
20.20. KernelRemap ..... 720
20.21. MergeComponents ..... 722
20.22. MergeKernel ..... 725
20.23. MergeParallel ..... 727
20.24. MergePixel ..... 730
20.25. NOP ..... 732
20.26. PARALLELdn ..... 734
20.27. PARALLELup ..... 737
20.28. PseudoRandomNumberGen ..... 740
20.29. SampleDn ..... 745
20.30. SampleUp ..... 748
20.31. SelectBitField ..... 750
20.32. SelectComponent ..... 752
20.33. SelectFromParallel ..... 754
20.34. SelectROI ..... 756
20.35. SelectSubKernel ..... 758
20.36. SetDimension ..... 760
20.37. SplitComponents ..... 762
20.38. SplitKernel ..... 765
20.39. SplitParallel ..... 766
20.40. Trash ..... 768
21. Library Blob ..... 769
21.1. Definition ..... 769
21.2. Definition of Object Features ..... 771
21.3. VisualApplets Operators ..... 775
21.4. Blob_Analysis_1D ..... 776
21.5. Blob_Analysis_2D ..... 791
22. Library Color ..... 799
22.1. BAYER3x3Linear ..... 800
22.2. BAYER5x5Linear ..... 803
22.3. ColorTransform ..... 807
22.4. HSI2RGB ..... 810
22.5. RGB2HSI ..... 812
22.6. RGB2YUV ..... 814
22.7. WhiteBalance ..... 816
22.8. WhiteBalanceBayer ..... 818
23. Library Compression ..... 820
23.1. ImageBuffer_JPEG_Gray ..... 821
23.2. JPEG_Encoder_Gray ..... 824
23.3. JPEG_Encoder ..... 830
24. Library Debugging ..... 836
24.1. ImageAnalyzer ..... 838
24.2. ImageStatistics ..... 844
24.3. StreamAnalyzer ..... 852
24.4. Scope ..... 858
24.5. ImageInjector ..... 862
24.6. ImageTimingGenerator ..... 866
24.7. ImageFlowControl ..... 872
24.8. StreamControl ..... 876
24.9. ImageMonitor ..... 879
25. Library Filter ..... 882
25.1. DILATE ..... 884
25.2. ERODE ..... 886
25.3. FIRkernelNxM ..... 888
25.4. FIRoperatorNxM ..... 894
25.5. HitOrMiss ..... 897
25.6. LineNeighboursNx1 ..... 899
25.7. MAX ..... 901
25.8. MEDIAN ..... 902
25.9. MIN ..... 903
25.10. NumberOfHits ..... 904

25.11. PixelNeighbours1xM ..... 906
25.12. SORT ..... 908
26. Library Logic ..... 909
26.1. AND ..... 911
26.2. CASE ..... 914
26.3. CMP_AgeB ..... 916
26.4. CMP_AgtB ..... 918
26.5. CMP_AleB ..... 920
26.6. CMP_AltB ..... 922
26.7. CMP_Equal ..... 924
26.8. CMP_NotEqual ..... 926
26.9. IF ..... 928
26.10. IS_Equal ..... 931
26.11. IS_GreaterEqual ..... 933
26.12. IS_GreaterThan ..... 935
26.13. IS_InRange ..... 937
26.14. IS_LessEqual ..... 939
26.15. IS_LessThan ..... 941
26.16. IS_NotEqual ..... 943
26.17. NOT ..... 945
26.18. OR ..... 947
26.19. XNOR ..... 949
26.20. XOR ..... 950
27. Library Memory ..... 951
27.1. CoefficientBuffer ..... 955
27.2. FrameBufferMultiRoiDyn ..... 962
27.3. FrameBufferRandomRead ..... 969
27.4. FrameBufferRandomRead (imaFlex) ..... 973
27.5. FrameMemory ..... 979
27.6. FrameMemoryRandomRd ..... 982
27.7. ImageBuffer ..... 985
27.8. ImageBufferMultiRoI ..... 990
27.9. ImageBufferMultiRoIDyn ..... 995
27.10. ImageBufferSC ..... 1000
27.11. ImageBufferSpatial ..... 1004
27.12. ImageFifo ..... 1008
27.13. ImageSequence ..... 1012
27.14. KneeLUT ..... 1015
27.15. LineBuffer (imaFlex) ..... 1022
27.16. LineMemory ..... 1027
27.17. LineMemoryRandomRd ..... 1031
27.18. LUT ..... 1034
27.19. RamLUT ..... 1037
27.20. RamLUT (imaFlex) ..... 1042
27.21. ROM ..... 1047
28. Library Parameters ..... 1049
28.1. EnumParamReference ..... 1063
28.2. EnumParamTranslator ..... 1068
28.3. EnumVariable ..... 1075
28.4. FloatFieldParamReference ..... 1078
28.5. FloatParamReference ..... 1084
28.6. FloatParamTranslator ..... 1089
28.7. FloatVariable ..... 1097
28.8. IntFieldParamReference ..... 1100
28.9. IntParamReference ..... 1105
28.10. IntParamTranslator ..... 1110
28.11. IntVariable ..... 1118
28.12. IntFieldVariable ..... 1121
28.13. LinkProperties ..... 1125
28.14. LinkParamTranslator ..... 1128
28.15. StringParamReference ..... 1135
28.16. ResourceReference ..... 1140

28.17. IntParamSelector ..... 1144
28.18. FloatParamSelector ..... 1148
29. Library Hardware Platform ..... 1152
29.1. AppletProperties ..... 1156
29.2. BoardStatus ..... 1164
29.3. ActionCommand ..... 1180
29.4. CameraControl ..... 1183
29.5. BaseGrayCamera ..... 1186
29.6. BaseRgbCamera ..... 1189
29.7. MediumGrayCamera ..... 1193
29.8. MediumRgbCamera ..... 1197
29.9. FullGrayCamera ..... 1203
29.10. FullRgbCamera ..... 1207
29.11. CameraGrayArea ..... 1212
29.12. CameraGrayAreaBase ..... 1214
29.13. CameraGrayAreaFull ..... 1216
29.14. CameraGrayAreaMedium ..... 1218
29.15. CameraGrayLine ..... 1220
29.16. CameraGrayLineBase ..... 1222
29.17. CameraGrayLineFull ..... 1224
29.18. CameraGrayLineMedium ..... 1226
29.19. CameraRgbArea ..... 1228
29.20. CameraRgbAreaBase ..... 1230
29.21. CameraRgbAreaMedium ..... 1232
29.22. CameraRgbLine ..... 1234
29.23. CameraRgbLineBase ..... 1236
29.24. CameraRgbLineMedium ..... 1238
29.25. CLHSDualCamera ..... 1240
29.26. CLHSPulseIn ..... 1244
29.27. CLHSPulseOut ..... 1248
29.28. CLHSSingleCamera ..... 1252
29.29. CxpCamera ..... 1257
29.30. CxpCameraMultiTap ..... 1265
29.31. CxpAcquisitionStatus ..... 1277
29.32. CxpPortStatus ..... 1278
29.33. CxpRxTrigger ..... 1288
29.34. CxpTxTrigger ..... 1290
29.36. CXPDualCamera ..... 1293
29.37. CXPQuadCamera ..... 1301
29.38. CXPSingleCamera ..... 1309
29.39. DigIOPort ..... 1317
29.40. DmaFromPC ..... 1318
29.41. DmaToPC ..... 1321
29.42. GPI ..... 1325
29.43. GPO ..... 1329
29.44. LED ..... 1333
29.45. NativeTrgPortIn ..... 1336
29.46. NativeTrgPortInExt ..... 1337
29.47. NativeTrgPortOut ..... 1338
29.48. RxLink ..... 1339
29.49. TrgPortArea ..... 1341
29.50. TrgPortLine ..... 1345
29.51. TriggerIn ..... 1356
29.52. TriggerOut ..... 1358
29.53. TxLink ..... 1360
29.54. SignalToEvent ..... 1362
30. Library Prototype ..... 1364
30.1. COUNTER ..... 1365
30.2. CustomSignalOperator ..... 1367
30.3. HWMULT ..... 1369
30.4. PackbitsRLE ..... 1371
30.5. TrgBoxLine ..... 1373

30.6. RGB2XYZ ..... 1392
30.7. XYZ2LAB ..... 1395
31. Library Signal ..... 1396
31.1. DelayToSignal ..... 1399
31.2. Downscale ..... 1402
31.3. EventToSignal ..... 1405
31.4. FrameEndToSignal ..... 1407
31.5. FrameStartToSignal ..... 1409
31.6. Generate ..... 1411
31.7. GetSignalStatus ..... 1417
31.8. Gnd ..... 1419
31.9. LimitSignalWidth ..... 1422
31.10. LineEndToSignal ..... 1425
31.11. LineStartToSignal ..... 1427
31.12. PeriodToSignal ..... 1429
31.13. PixelToSignal ..... 1432
31.14. Polarity ..... 1434
31.15. PulseCounter ..... 1437
31.16. RsFlipFlop ..... 1439
31.17. RxSignalLink ..... 1441
31.18. Select ..... 1443
31.19. SetSignalStatus ..... 1446
31.20. ShaftEncoder ..... 1449
31.21. ShaftEncoderCompensate ..... 1453
31.22. SignalDebounce ..... 1456
31.23. SignalDelay ..... 1459
31.24. SignalEdge ..... 1462
31.25. SignalGate ..... 1464
31.26. SignalToDelay ..... 1469
31.27. SignalToPeriod ..... 1471
31.28. SignalToPixel ..... 1473
31.29. SignalToWidth ..... 1475
31.30. SignalWidth ..... 1477
31.31. SyncSignal ..... 1481
31.32. TxSignalLink ..... 1483
31.33. Vcc ..... 1485
31.34. WidthToSignal ..... 1488
32. Library Synchronization ..... 1491
32.1. AppendImage ..... 1494
32.2. AppendImageDyn ..... 1497
32.3. AppendLine ..... 1499
32.4. AppendLineDyn ..... 1501
32.5. CutImage ..... 1503
32.6. CutLine ..... 1505
32.7. CreateBlankImage ..... 1507
32.8. ExpandLine ..... 1510
32.9. ExpandPixel ..... 1512
32.10. ImageValve ..... 1514
32.11. InsertImage ..... 1516
32.12. InsertLine ..... 1519
32.13. InsertPixel ..... 1522
32.14. IsFirstPixel ..... 1524
32.15. IsLastPixel ..... 1526
32.16. PixelReplicator ..... 1531
32.17. PixelToImage ..... 1533
32.18. RemoveImage ..... 1536
32.19. RemoveLine ..... 1538
32.20. RemovePixel ..... 1540
32.21. ReSyncToLine ..... 1545
32.22. RxImageLink ..... 1547
32.23. SourceSelector ..... 1550
32.24. SplitImage ..... 1552

32.25. SplitLine ..... 1556
32.26. SYNC ..... 1558
32.27. TxImageLink ..... 1571
32.28. Overflow ..... 1574
33. Library Transformation ..... 1576
33.1. FFT ..... 1577
Appendix A. Device Resources ..... 1580
A.1. Hardware Configuration of Supported Platforms ..... 1580
A.2. Device Resources of Supported Platforms ..... 1583
A.3. Shared Memory Concept ..... 1586
Glossary ..... 1589
Bibliography ..... 1592
Index ..... 1593

# Part I

User Manual

# 1. Introduction

## 1.1. VisualApplets




Welcome to the world of modern FPGA programming. By purchasing VisualApplets you own one of the leading and most advanced tools for FPGA programming to realize image processing applications.

VisualApplets will:

- turn the frame grabber or camera into a flexible and intelligent high-performance image processor
- let you deploy the potential of modern FPGA technology at any customer
- enable you to realize real-time solutions for applications with image processing requirements in minutes
- upgrade your application to a reliable hardware solution level

VisualApplets is a hardware programming tool for FPGAs, based on the use of graphical pipeline-structure objects.

Image processing designs are arranged by the combination of operator modules, filter modules and transport links. The provided libraries contain more than 200 hardware-based operators which cover standard as well as advanced image processing functions.

Included in delivery are arithmetical and morphological operators for pixel manipulation, logical operators for classification tasks, complex modules for color processing, operators for statistics analysis and processing of image sequences. Additional operators are responsible for format conversion, compression or conversion in pixel lists. Special features are the programming of the control signals to individualize the trigger functionality, and the segmentation and classification functions in the blob analysis operator. Complex operators can be combined by basic ones and stored in individual user libraries. This allows building complex designs without losing clarity of the design.

The complete set of functions is implemented as hardware operators and guarantees image processing in real-time. The complexity of image processing designs is mainly limited by the available resources of the FPGA hardware. There is no need for a user to waste time debugging synchronization issues, manually managing bandwidth and timing. Also, a build function and high level simulation are integrated to offer full control over the final visual result of a design on bit accuracy from within the development environment.

The complete process of the hardware design creation can be completed in a matter of minutes. The hardware applet can immediately be verified by microDisplay, the viewer and configuration software, or be integrated in applications by use of the pre-generated SDK example code.

Although knowledge of hardware programming is advantageous, the software solution VisualApplets® is addressed to application engineers in Machine Vision as a matter of priority.

VisualApplets is target hardware independent.

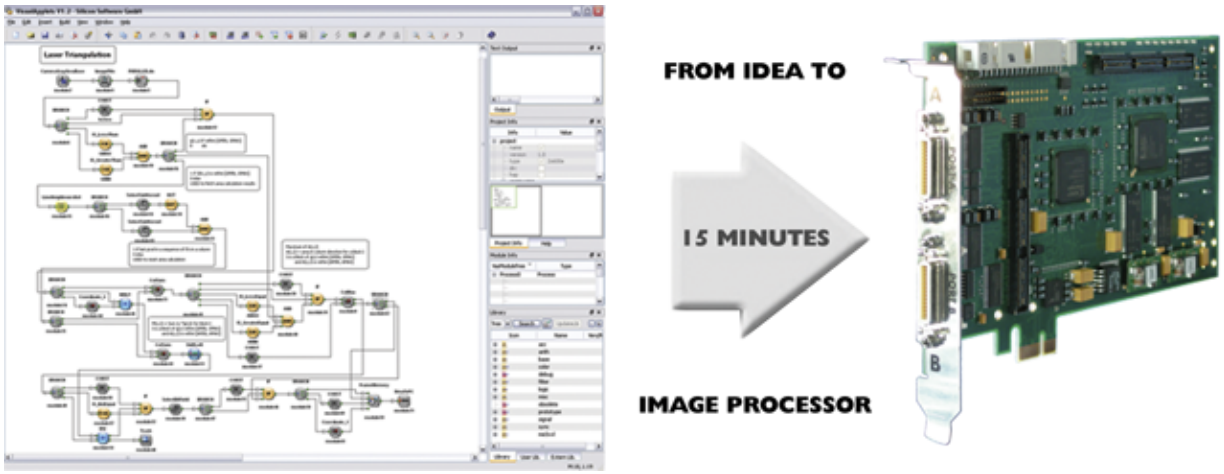



Figure 1.1. VisualApplets - From Idea to Image Processor in 15 Minutes

VisualApplets's key benefits and features:

- a graphical interface to program FPGA hardware
- no knowledge required of circuitry, synchronization, timings or FPGA programming
- libraries with hardware modules representing available hardware configurations
- image processing libraries with various filter operators and imaging modules
- availability of VisualApplets® image processing libraries with special market, branch or customer focus
- build and high-level simulation in software
- no need for a VHDL compiler
- accessible to hardware developers and application engineers alike
- closes the gap between standard and custom specific applets




Figure 1.2. VisualApplets – Awarded Software Environment

Basler hopes you will enjoy the “world of VisualApplets” and wish a successful time in developing your individual real-time applications.

## 1.2. How to Use This Documentation

This documentation is divided into three major parts.

The **Part I, 'User Manual'** lists and explains all functions of VisualApplets. If you are new to VisualApplets, we recommend you start with Section 2.1, ' Writing Your First Applet '. To get an overview over the range of functionalities VisualApplets offers, proceed with 3. *Basic Functionality* and 4. *Extended Functionality*. All information provided by the User Manual is additionally available directly in the program as context-sensitive online help.

Part II, 'Tutorial and Examples' provides a deeper step-by-step introduction into VisualApplets.

In the Part III, 'Operator Reference' , you find a complete and detailed description of all operators.



**New Layout of Operator Icons**

The layout of the operator icons has been improved, so that the operator type (O or M type) is very easy to recognize now. Some screenshots in this document might not yet reflect these changes, but this will not have any impact on the clearness and comprehensibility of this documentation.

## 1.3. System Requirements

System requirements, see Installing VisualApplets [https://docs.baslerweb.com/visualapplets/installing-visualapplets]

# 2. Getting Started

## 2.1. Writing Your First Applet

To get a first impression on VisualApplets, this chapter gives you a short introduction into the tool. All steps required to generate an image processing application with VisualApplets are presented. You will learn how fast applications can be realized and how easy it is without the need on FPGA or any other hardware-specific knowledge.

### 2.1.1. Designing the Applet

1. Start VisualApplets by clicking on the VisualApplets program icon in the Windows **Start** menu or on the Desktop. At first, the main window will open with no project loaded.

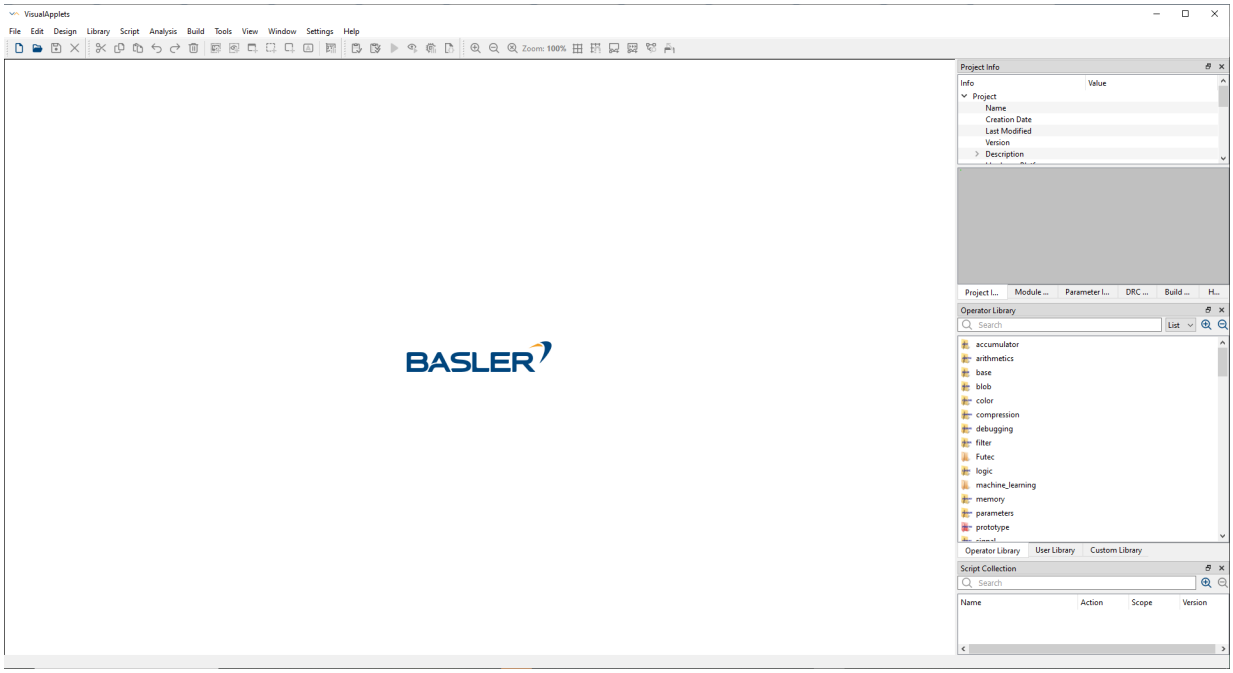



Figure 2.1. VisualApplets Main Window

You can now immediately start implementing your first design. The first step will be to start a new project.

2. Click on **File** -> **New** ( **Ctrl+N** ) or use the *New* icon from the *File* icon bar. A *New Project* window will open which allows you to specify project name, target hardware platform, and target runtime. If you don't know these settings at the current step of development, just give a name to your project and use the default settings. You can always change these settings later on.

3. To follow our example here, just use the following settings:

   - Project Name: "helloWorld"
   - Hardware Platform: microEnable IV VD4-CL frame grabber
   - Target Runtime: Microsoft Windows 64-bit system.

4. Confirm your settings by clicking on **OK**.

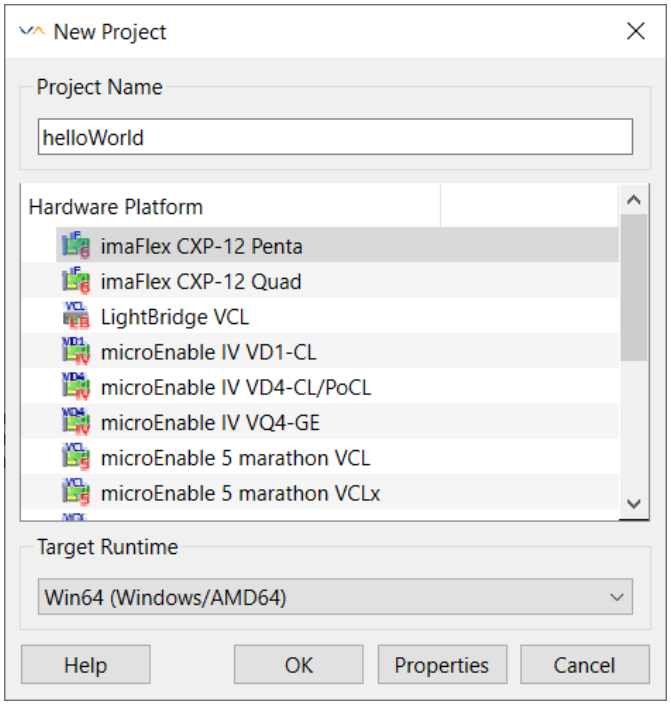



Figure 2.2. Start of a New Project

VisualApplets will now start a new project. You will see a blank design window in the center of your program window. (If you selected microEnable 5 as hardware platform, the two obligatory control operators *Applet Properties* and *Board Status* are automatically inserted into the empty design.) In the *Project Info* tab on the right, information regarding your current project like project name, target hardware, target platform etc. is displayed.

In VisualApplets, image processing operations are represented by operators. All these operators can be found in the operator library on the right of the VisualApplets design window. Using drag-and-drop, you can very easily place operators into the design window. An instance of an operator in the design is called a **module**. Operators can have input and output ports. Operators in a design (i.e., modules) can be connected using these ports. Connections between modules are called **links** which are represented in the design window by arrows. These modules and links represent the image- (or signal-) processing pipeline; hence, the order of operations is determined by the order of modules.

For our first design, we will need the following three operators:

- Operator *CameraGrayAreaBase* from the *mE4VD4-CL* library
- Operator *ImageBuffer* from the *Memory* library
- Operator *DmaToPC* from the *mE4VD4-CL* library

To use these operators in your design:

5. Locate the operators in their libraries and drag them into your design window as shown in the following:

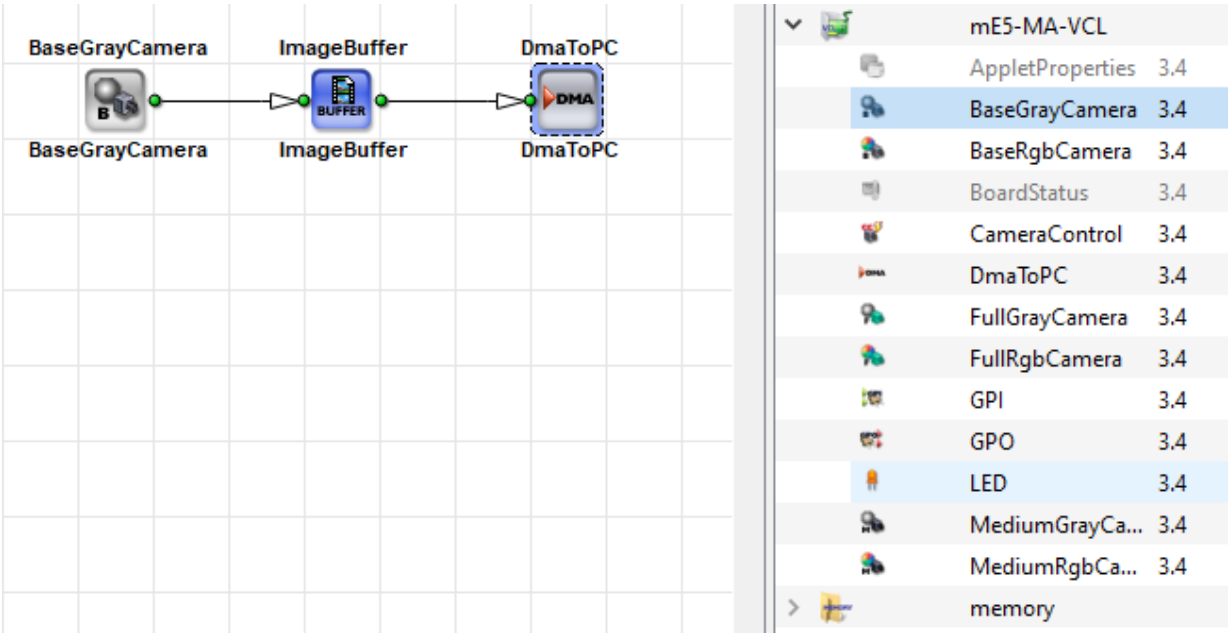



Figure 2.3. Dragging Operators from Libraries into the Design Window

The next step will be to connect the modules in your design via links.

Clicking on a port will start a new link which can be finalized with a second click on the target port.

6. To connect the camera module with the buffer module, click on the output port of the camera module and then on the input port of the buffer module.
7. Connect the buffer module with the DMA module in the same way.

   Save your Design. If you save your design for the first time, the *Save* dialog will offer the project name as the name for your design file.

**File Name and Project Name**

If you save your design for the first time, or if you use the *Save as...* option, the *Save*/*Save as...* dialog will offer the project name as the name for your design file.

## 2.1.2. Parametrizing

Now you have finished your first design. In the next step, you have to specify the settings of the individual modules and links (otherwise, the design will use the preset default properties).

1. To parametrize a module, just double-click on the module. A dialog window will open where all parameters of the module are listed.
2. Set the properties as you need them for your design. For this first sample applet you are designing right now, simply change the names of your modules to "Camera", "Buffer", and "DMA".
3. Click on **Apply**.
4. Close the properties windows by clicking on **Close**.
5. Save your Design.

**Operator Parameters**

Parameters are always operator-specific; therefore, the setting options differ from operator to operator.

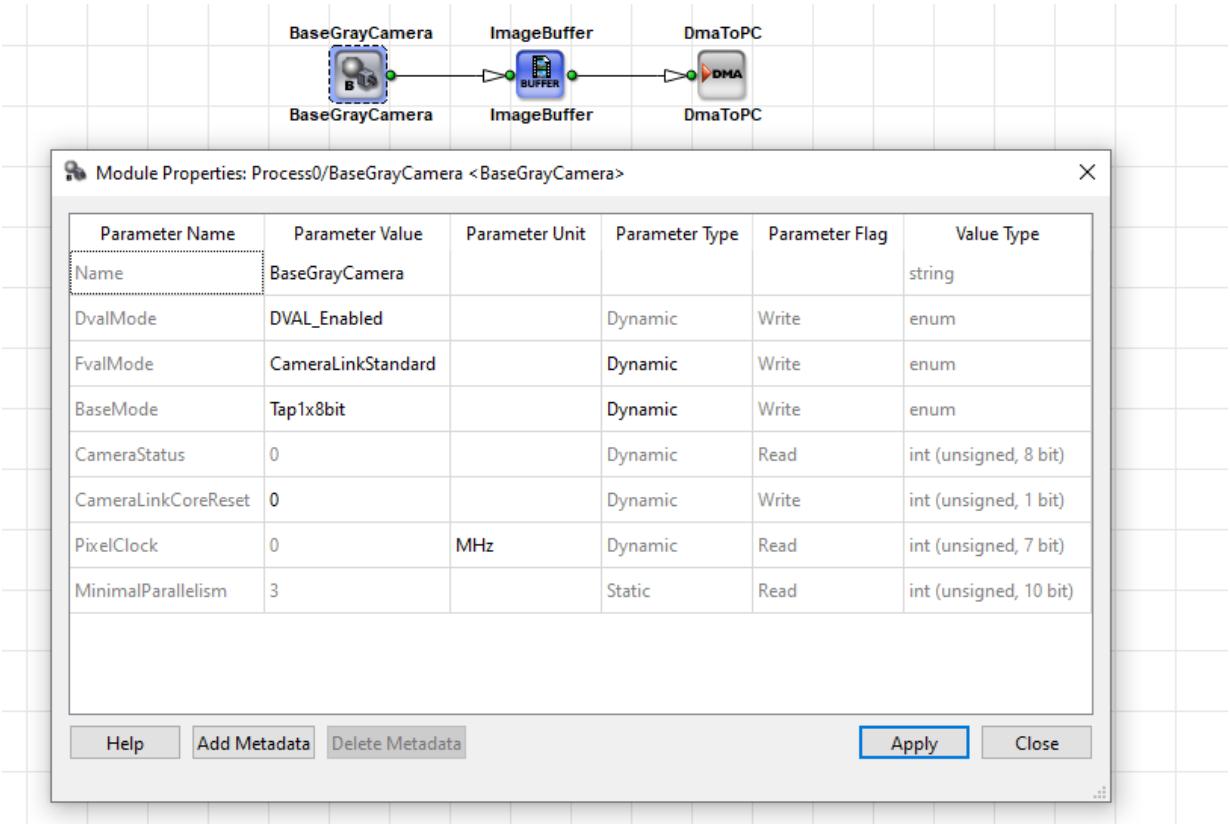



Figure 2.4. Module Properties

The next step is to edit the properties of the links of your design. In the sample applet you are creating just now, there is no need to change any of the link properties. You can simply use the default settings.

However, whenever you need to parametrize the links of your design, you should proceed as follows:

6. Double-click on the link you want to parametrize.
7. Enter the desired settings.
8. Click on **Apply**.
9. Close the properties window by clicking on **Close**.

## 2.1.3. Design Rules Check (DRC)

You have now fully implemented your first application which is a simple image acquisition. Proceed by checking your design for errors using the Design Rules Check (DRC) functions of VisualApplets.

1. Click on **Analysis** -> **Design Rules Check Level 1** ( **Ctrl+F7** ) or use the icon Design Rules Check Level 1 from the Build icon bar.
2. In the *Project Info* window on the right, change to the *DRC Log* tab.

Here, The DRC analysis result is displayed as you can see in the following figure:

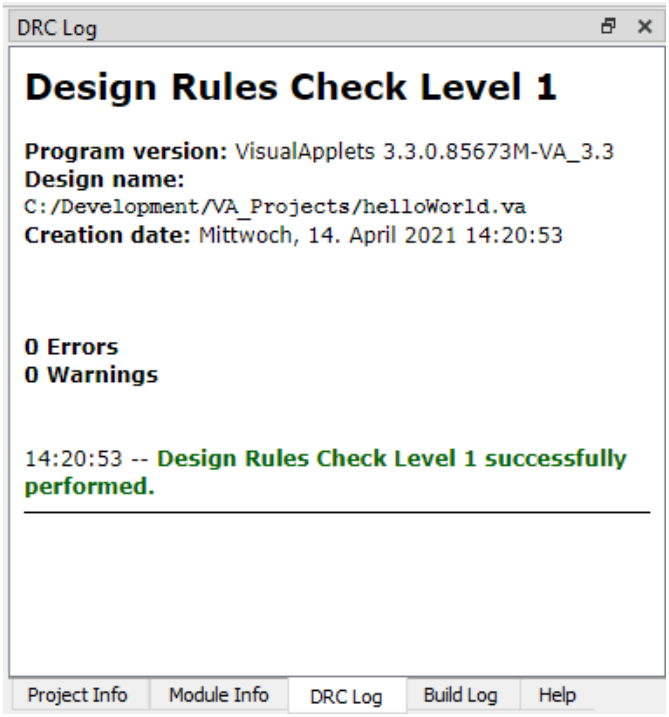



Figure 2.5. Successful DRC

**Important**

If the DRC detects an error, you have to correct it before continuing with the next step. Detected errors are listed in link format. If you click on one of the listed errors, the respective module or link will be highlighted in the Design Window.

After successful DRC, save your design:

3. Click on **File** -> **Save** ( **Ctrl+S** ) or use the *Save* icon from the *File* icon bar.

### 2.1.4. Editing the Build Settings

You need to edit the build settings, if

- You create designs for a microEnable 5 (ironman and marathon) or LightBridge platform.
- You work with Xilinx Vivado.
- You work with a Xilinx ISE version higher than 9.2.

You can skip this section and proceed with Section 2.1.5, 'Building the Final Hardware Applet', if

- you design for microEnable IV frame grabbers and use Xilinx ISE 9.2 for building these designs.

**mE 4 Users**

If you are designing for a microEnable IV platform and use a Xilinx ISE version 9.2 or lower, skip this section and proceed with Section 2.1.5, 'Building the Final Hardware Applet'.

To set the build settings for your specific environment, proceed as follows:

1. Click menu **Settings** -> **Build Settings**.

   The *Build Settings* dialog opens.

2. Click on the **Add** button.
3. Select the target hardware platform (frame grabber) for these build settings on and confirm with **OK**.
4. Give a name to the new set of build settings you are just creating.
5. Activate the option *Active configuration* (directly behind the filed where you entered the name).
6. Leave *Precondition Check* activated.
7. Under *Xilinx Build Flow*, select the Xilinx Tool you want to use (Xilinx ISE or Xilinx Vivado).
8. Disable the option *Use system environment instead*.
9. Select a Xilinx settings batch file from your file system. You find the batch file in the Xilinx installation folder.:
   - **ISE:** \Xilinx\<version_number>\ISE_DS\settings64.bat.
   - **Vivado:** \Xilinx\Vivado\<version_number>\settings64.bat.

   We recommend to use the 64-bit Windows operating system when developing applets for microEnable 5 platforms. Make sure you select the batch file that matches the operating system you are using, e.g., "settings64.bat" which is the file for the 64-bit Windows OP.
10. Keep all other settings as they are.
11. Click *OK*.

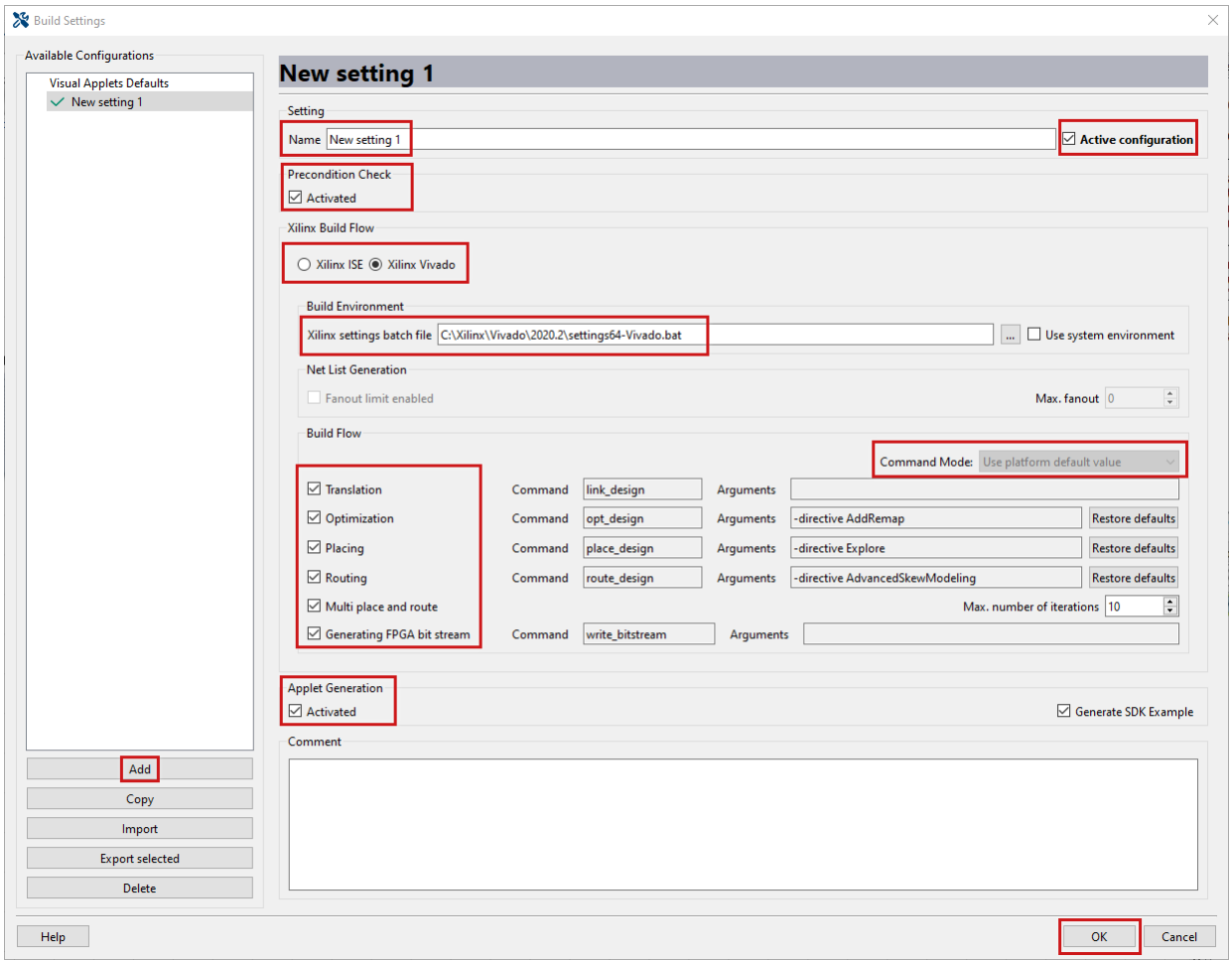



Figure 2.6. Build Settings for microEnable 5 / Xilinx Vivado

For further details on build settings options, refer to section Section 4.9, 'Build Settings'

### 2.1.5. Building the Final Hardware Applet

The last step is the build of your design.



**Build Preconditions**

The build process can only be performed if the XILINX tools are properly installed. Refer to section Installing the Xilinx Toolchain [https://docs.baslerweb.com/visualapplets/installing-visualapplets#installing-the-xilinx-toolchain] for more information on XILINX tool installation.

If you create designs for microEnable 5 (ironman and marathon) or LightBridge frame grabbers, or if you work with a Xilinx ISE version higher than version 9.2, you need to define the build settings before proceeding, see section Section 2.1.4, 'Editing the Build Settings '.

To create the bit stream (*.hap file):

1. Click on **Build** -> **Build** ( **F7** ) or use the Build icon from the *Build* toolbar.

   The *Build* dialog opens.

2. Under *Xilinx configuration*, select the build configuration you want to use (out of the configurations you defined in the *Build Settings* dialog).

3. Click on **Start** to actually start the bitstream generation.

VisualApplets will now use the XILINX tools to translate the application into the FPGA bitstream, i.e., the "program" or "applet".



**Prolonged Build Time Possible**

The duration of the Place and Route process depends on the complexity of the design.

The build of highly complex designs might take several hours.

After successful build, the applet is fully generated.

The name of your applet (*.hap file) is the same as the name of the design file (*.va). The project name has no influence on the name of the applet (*.hap file).

At this point, you are done with VisualApplets and can use the applet in real FPGA hardware.

An explanation on how to run a VisualApplets design on hardware you will find in section Section 2.2, 'Running Your Applet on Hardware'.

## 2.2. Running Your Applet on Hardware



**Prerequisites:**

- A programmable (V Series) microEnable frame grabber (hardware) is installed on your system.
- The Framegrabber SDK is installed on your system.

How you get your applet running depends on the frame grabber generation you are using.

**microEnable 5 (marathon and ironman) and LightBridge:** With microEnable 5 and LightBridge frame grabbers, you first have to load your *.hap file (applet) via firmware flasher tool onto your frame grabber, prior to selecting and starting the applet (see Section 2.2.1, 'Flashing').

**microEnable IV:** With microEnable IV, you can immediately select the applet (*.hap file) from your local file system and start it on the frame grabber.

You can

- directly see and test the image processing results of your applet with the tool microDiagnostics (which comes as part of the Framegrabber SDK) (see https://docs.baslerweb.com/frame-grabbers/managing-applets-micro-diagnostics).
- start the applet via the application programming interface (see Section 2.2.3, 'Starting the Applet in your own Software').

### 2.2.1. Flashing

You need to flash your frame grabber with the new applet, if you use a microEnable 5 (marathon, ironman) or a LightBridge frame grabber.

If you are going to use the new applet on a microEnable IV frame grabber, just skip this section.

To flash your frame grabber:

1. Start the tool microDiagnostics.
2. Select the frame grabber you want to use.
3. Click the button **Flash Selected Board(s)**. A new window opens.
4. Go to the directory where you store your *.hap file created in VisualApplets, and select the file.
5. Click on **Open** and confirm by clicking on **Yes**.
6. Wait until the new firmware is completely installed. You get an according message in microDiagnostics.
7. Follow the instructions in the message.

For detailed information how to flash your specific frame grabber, refer to the User Manual of your specific frame grabber (Flashing Applets onto marathon Frame Grabbers [https://docs.baslerweb.com/frame-grabbers/managing-applets-micro-diagnostics#flashing-applets-onto-marathon-frame-grabbers]).

### 2.2.2. Testing your Applet in microDisplay

To test your applet and to set some first parameters, you can use the tool microDisplay:

1. Save the `*.hap` file you created in VisualApplets into the Framegrabber SDK installation directory:

   `[Framegrabber SDK installation directory]/Hardware Applets`.

2. Start the tool microDisplay, either directly from VisualApplets by clicking on **Build** -> **microDisplay** ( **F5** ), or via Windows START menu.
3. In the dialog *I want to…*, select *Load Applet*.
4. In the *Load Hardware Applet* dialog, select the frame grabber you want to use under *Board*. Immediately, all applets available for the selected board are displayed.

   With microEnable 5, only one applet (the one you flashed your board with) will be available.

5. Select the `*.hap` file you want to use.

6. Click on the *Load* button to load the selected applet (`*.hap` file) onto the frame grabber.
7. Close the *Load Hardware Applet* dialog.
8. Configure the frame grabber using the parameter panel of the microDisplay program window.

   When working with more than one camera, first select the port you are going to configure. Configure all ports you are using.

   Set the parameters to your needs, e.g., image height and image width. To do so, right-click directly on the value and select *Edit*.

   Configure specific operation modes you want to use (e.g., trigger settings).
9. Start image acquisition on the frame grabber by clicking on the button *Grab and display an infinite number of frames*.

   The grabbed images are now displayed in microDisplay.
10. To stop the acquisition, click on the *Stop* button in microDisplay.

### 2.2.3. Starting the Applet in your own Software

To start using the Applet in your own software:

1. In the Framegrabber API, use the call `Fg_Init` to start the applet.

   Specify the path to the location of the `*.hap` file you created with VisualApplets.



**Starting the Camera**

Depending on the type of camera interface, it might be helpful to start GenICam explorer tools as well in order to control the camera. For details, please refer to the Framegrabber SDK documentation [https://docs.baslerweb.com/frame-grabbers/configuring-the-camera-micro-display-x ].

## 2.3. Further Reading

You have just successfully implemented your first VisualApplets design. If you are interested in a deeper step-by-step introduction into VisualApplets we recommend to read our tutorial which you can find in Part II, 'Tutorial and Examples'.

If you want to learn more about the functionalities of the program, just proceed by reading the next chapters of this user manual in Part I, 'User Manual'. Here you will find all functions of VisualApplets explained in detail.

All information provided in the User Manual you can also access directly in VisualApplets. Click on a **Help** button in the program and you will be automatically directed to the respective explanations in the user manual.

If you prefer a directed tool introduction, Basler offers VisualApplets workshops and coaching. Feel free to contact us via Basler Sales [https://www.baslerweb.com/en/sales-support/sales/].

# 3. Basic Functionality

This chapter outlines the fundamental ideas and functions of VisualApplets. First, you get an overview over the basic principles of VisualApplets, the typical workflow of the design process, and the graphical user interface you will use for creating your designs. Afterwards (starting with Section 3.4, 'Entering a Design'), all functions vital to generate VisualApplets designs are explained in detail. This includes operator insertion, module and link parametrization, verification, simulation and, finally, the build process.

A detailed description of the extended functions of VisualApplets you will find in 4. *Extended Functionality*. If you prefer a step-by-step introduction into the program, we recommend reading Part II, 'Tutorial and Examples' in detail.

## 3.1. Basic Principles

The basic principles of VisualApplets are very simple.

- Image processing or signal processing functions are represented by abstract operators.
- Operators are selected in accordance with the intended functionality of the applet. They can be chosen from extensive libraries.
- The operators are inserted into the user's design. The instance of an operator in a design is called *module*.
- A data flow model which represents the processing chain is build by linking modules.
- All modules and links can be parametrized to meet the requirements of the user's application and make the implementation fast and efficient.
- After successful offline verification, the design is built into a hardware applet file which can be loaded onto the hardware devices.

The following figure shows a screen shot of a simple VisualApplets design. As you can see, the modules in the design window are combined via links which are represented by arrows. The order of the modules define the functionality of the final applet. In this example, we acquire images from a camera (module *Camera*), buffer the images in an on-board memory (*Buffer*), and output the result to the host PC using the operator *DmaToPC*.

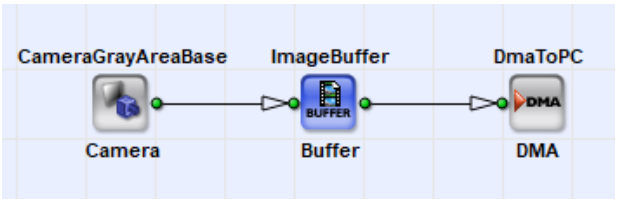



Figure 3.1. Simple VisualApplets Design

After all operators have been placed into a design, have been connected using links and parametrized, the design has to be verified. The program offers multiple functionalities to check consistency, bandwidth, and resources. One very important part of the verification is the functional simulation. Here, the design can be simulated with data from image files without the need of any target hardware.

Finally, the design is build to become a hardware applet file (HAP) which can be used on the target hardware devices. The next section will outline the workflow in detail.

## 3.2. Workflow

The required steps to generate or modify a VisualApplets project are the same for each project:

1. Design Entry:

   The design is build by using operators from the operator libraries and inserting them into the design. This way, the operators become instanciated. The instance of an operator within a design is called module. The modules of a design are connected via links. The order of the operators determine the functionality of design.

2. Design Parameterization:

   All modules and links of a design can be parameterized to define correct behavior, processing speed, and resource efficiency.

3. Design Verification:

   The verification of the design is a very important part of the design process. Two levels of design rule checks (DRC) verify the consistency of the design and show errors in parameterization and link formats, or resource conflicts.

   A functional simulation is used to verify the behavior of the implementation. Image files serve as source data. The simulation is performed completely offline, without the need of any target hardware.

4. Build:

   During the build process, the design is transformed into the binary hardware applet file (HAP) which can be directly loaded onto the hardware.

5. SDK Generator:

   To ease the integration of the applet into user applications, VisualApplets includes an SDK generator tool which generates a C++ project. The project can be immediately loaded and compiled and will provide examples on how to use and parameterize the new applet.

6. Applet Run:

   The applet can now be included and used in your application. To run applets, the Framegrabber SDK must be installed.

Figure 3.2, 'The Design Workflow' illustrates the steps described above. As you can see, during the design time of the applet, a *.va file is edited. After the build process, the final *.hap file is integrated into the user application.

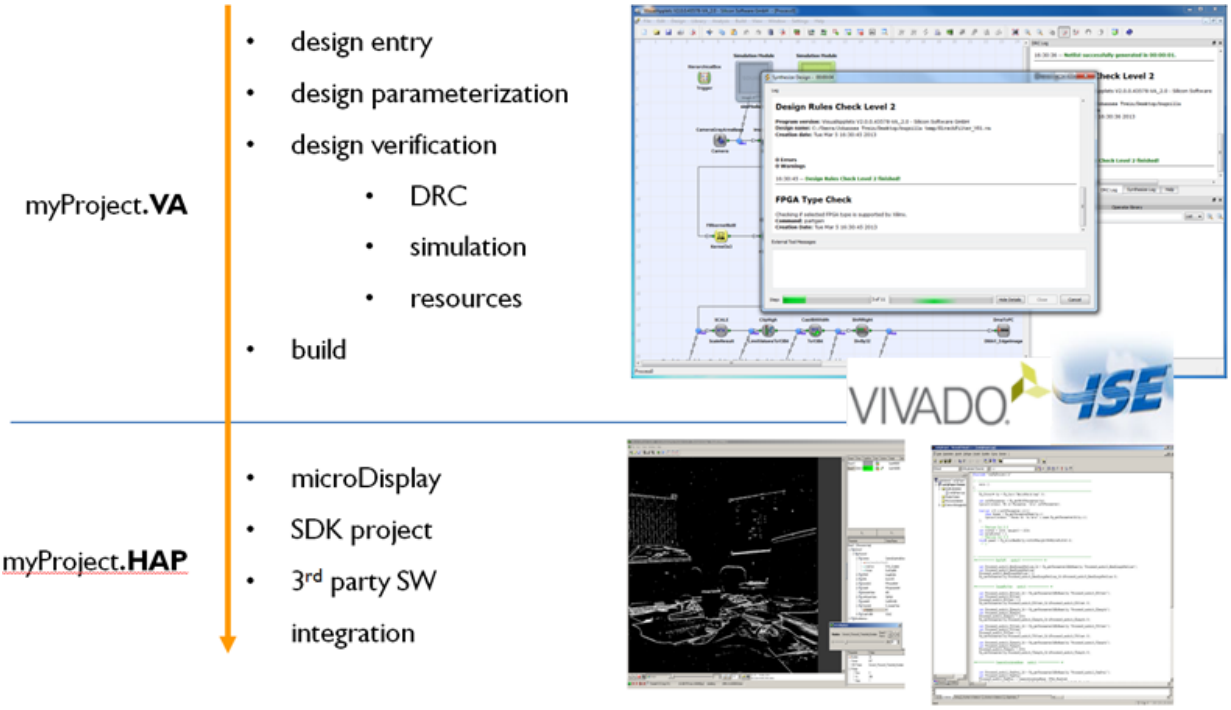



Figure 3.2. The Design Workflow

## 3.3. Main Program Window

VisualApplets offers a graphical user interface for applet design. No scripting, no programming code is required to be edited. The user interface is very simple and especially designed for easy learning by users. No technical background in programming is required.

The VisualApplets main program window consists of several bars and panels:

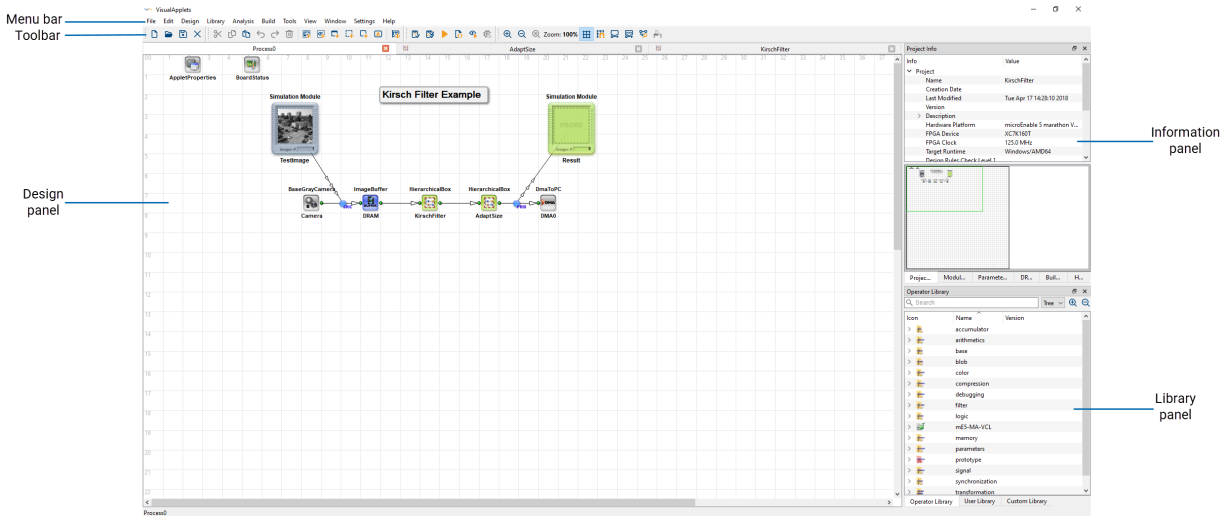
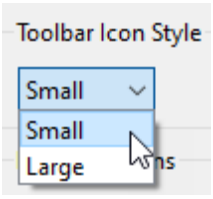



Figure 3.3. Main Program Window

The main menu bar provides direct access to the program features. Use the toolbar buttons for quick access to important functions. You can change the size of the toolbar icons in **Settings** -> **System Settings** -> **Common**:




Three panels provide different information and possibilities:

- Design panel: The design panel shows your VisualApplets project. You always have one main design window, but, depending on the structure of your project, you may additionally work with several sub windows. In the **Window** menu, multiple views can be selected (Tile, Cascade, Tabbed View). Using the menu entries you find in **Window** -> **Diagram Windows** ( **Ctrl+F8** ), you can easily select the design diagram window you want to work on at the moment.
- Information panel: In default setting of the main program window, the information panel is always displayed. It offers different information windows in tabbed view:
  - Project Info
  - Module Info
  - Parameter Info
  - DRC Log
  - Build Log
  - Help

  (For details on each information window, see below.)

- Library panel: In default setting of the main program window, the library panel is always displayed. It offers three library windows in tabbed view:
  - Operator Library: The Operator Library is a collection of sublibraries that contain VisualApplets operators, i.e. the individual image processing functions.
  - User Library: The User Library can include custom libraries with user-defined combinations of operators (hierarchical boxes). For more information see Section 4.2, 'User Libraries'.
  - Custom Library: To import operators into the Custom Library you need either an **Expert** license or the **VisualApplets 4** license.



**Specific Operators**

Some operators displayed in the operator library may not be available for the platform you are currently designing for. If not available for the platform you selected, the operator is grayed-out.

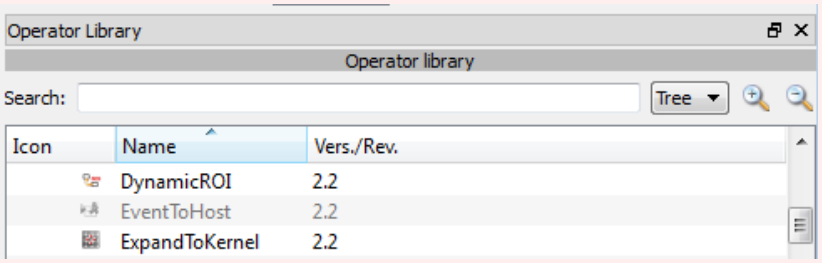



Figure 3.4. Operator not available for currently selected target hardware platform

## 3.3.1. The Windows of the Information Panel

### 3.3.1.1. Project Info

The Project Info window summarizes information on the current project.

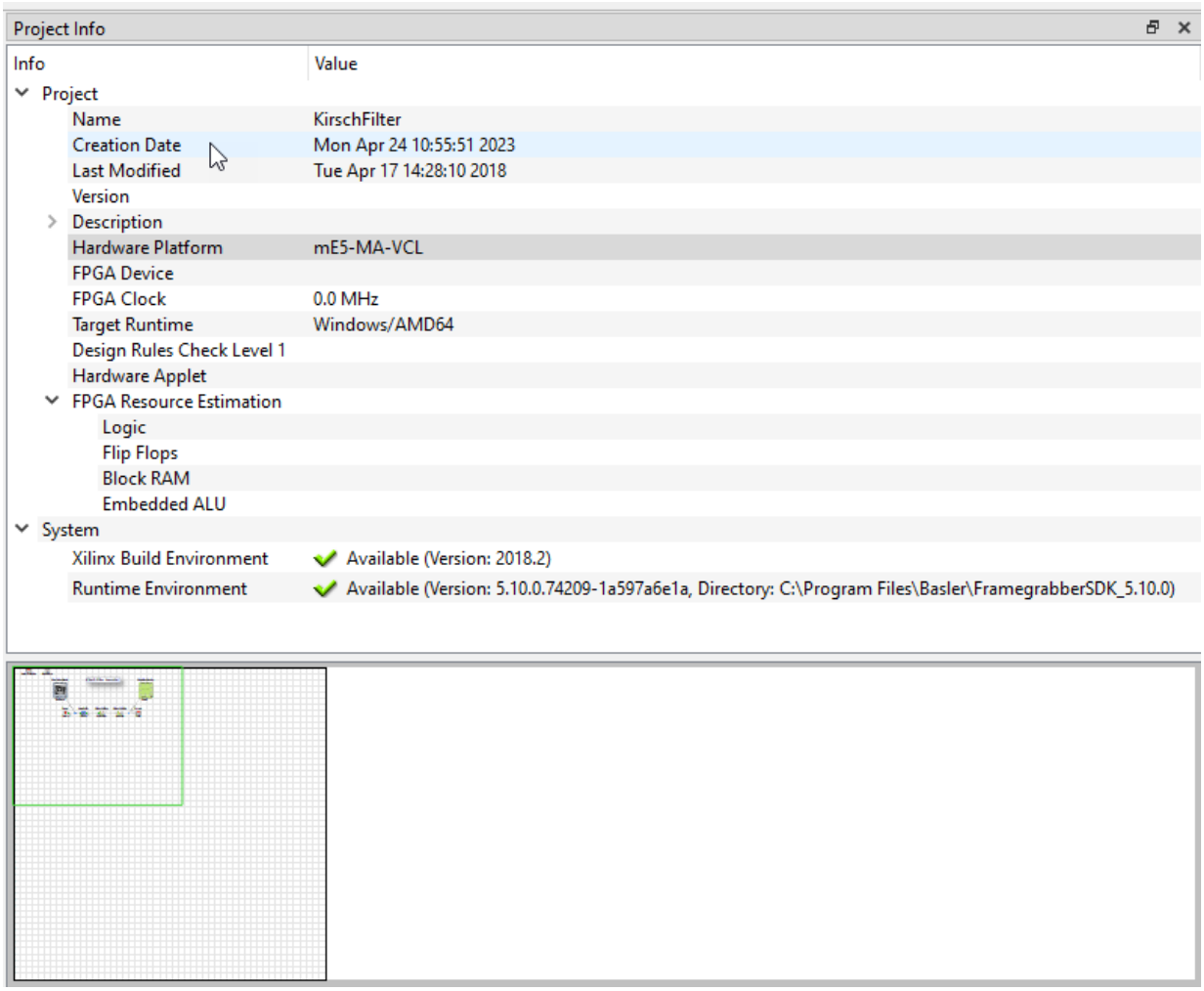



Figure 3.5. Project Info

The following information is provided:

- Name: Name of project (=file name)
- Creation Date: Date of creation
- Last Modified: Date of last modification
- Version: Version number of the project; this entry can be edited from the **Design** -> **Project Details** menu (see Section 4.8.2, 'Project Properties')
- Description: Verbal description of the project; this entry can be edited from the **Design** -> **Project Details** menu (see Section 4.8.2, 'Project Properties')
- Type: Displays the FPGA type currently used on the target hardware (see Section 4.5, 'Target Hardware Porting')
- Target Runtime: Displays the target runtime (see Section 4.8.1, 'Target Runtime')
- Design Rules Check Level 1: Indicates found errors or successful performance (see Section 3.9, 'Design Rules Check')
- Hardware Applet: Indicates the existence of a hardware applet. The entry is checked if VisualApplets previously generated a HAP file of the project.
- Resources: Shows the results of the resource estimation. See also Section 3.11, 'FPGA Resource Estimation'.

- System: Displays the found XILINX version, the available Framegrabber SDK version, and the Framegrabber SDK installation directory. This information is used by VisualApplets to build a HAP and for the final HAP output. See Section 3.12, 'Build' for more information.

### 3.3.1.2. Module Info

The *Module Info* window shows a list of all modules used in the project (as long als the input field of the window is empty). The list is structured by the hierarchy of the project.

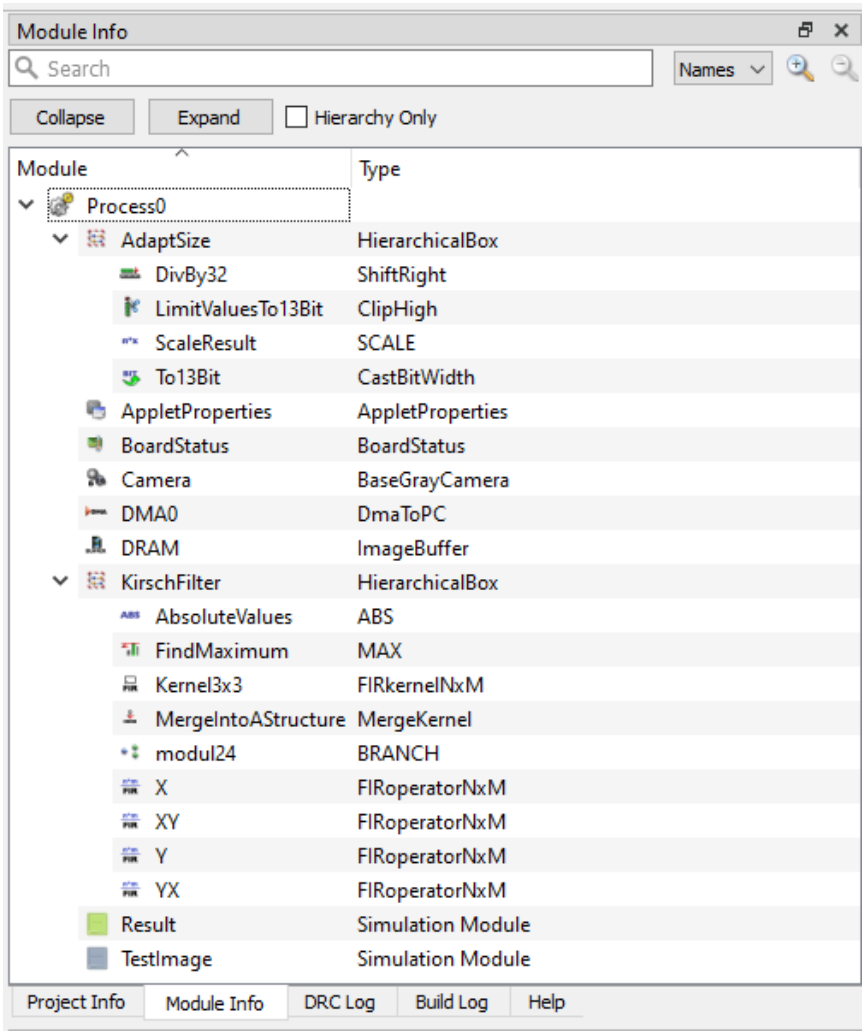



Figure 3.6. Module Search

In the Module Info window, you can easily search for a certain module or module type. This is very helpful when working in large designs. Before you start your search, you should define the filter settings:

- If you want to search for the individual name you gave to the module, activate "Names".
- If you want to search for the module type (the name of the operator the instance of which you are currently searching for), activate "Types".

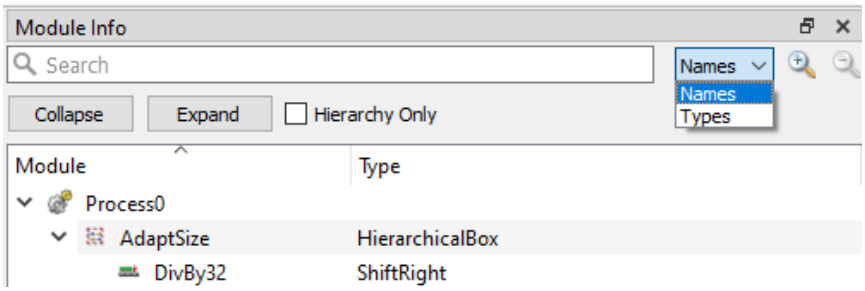



Figure 3.7. Module Search

You can now enter your search string. It is sufficient to enter only part of a name. A click on one of the modules listed in the *Module Info* window will bring the respective design window into focus (in the design panel), with the module selected and highlighted.

### 3.3.1.3. Parameter Info

The *Parameter Info* view shows all parameters and their operators of the active design. You can also edit the parameters in the *Parameter Info* view.

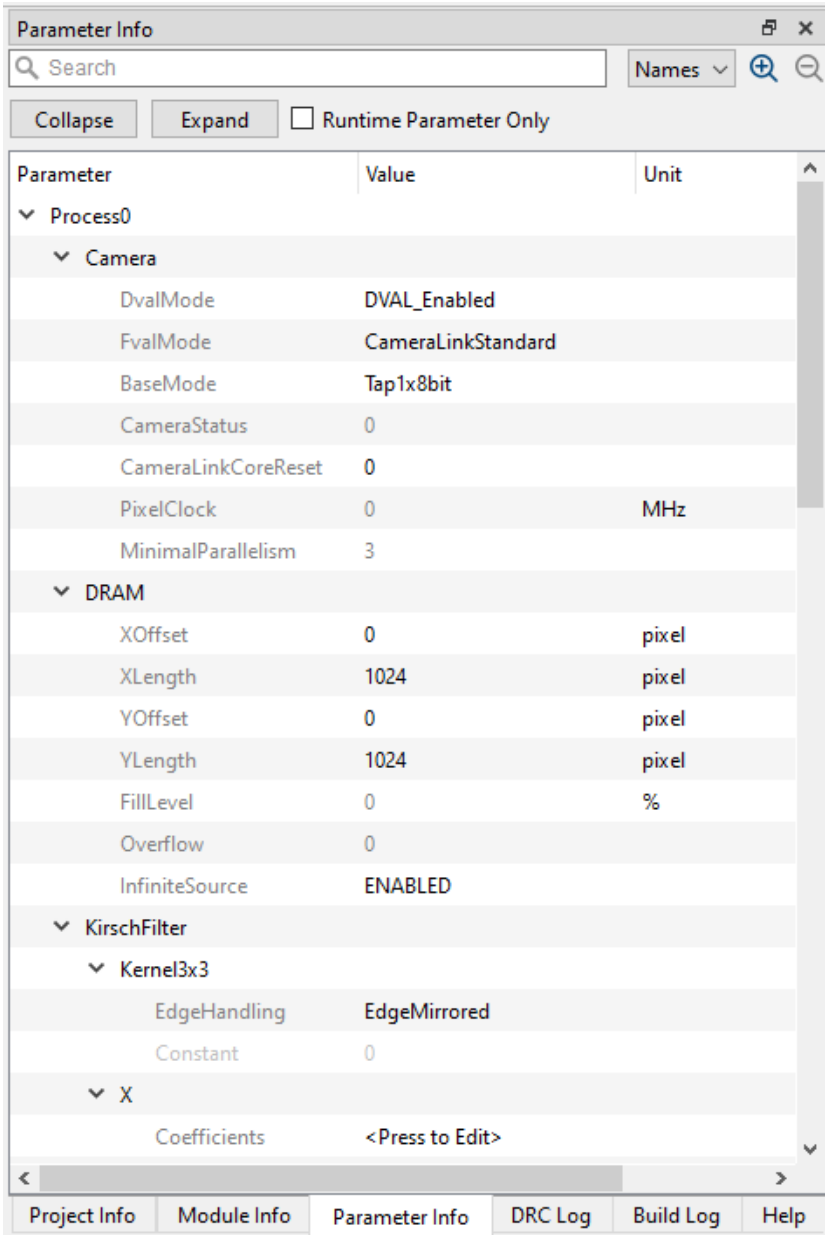
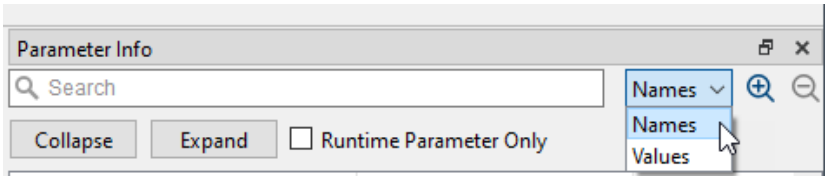



Figure 3.8. Parameter Info

The *Parameter Info* view provides a search function that allows you to search for parameters, parameter values, or operator names. You can use the *Runtime Parameter Only* filter to display only the dynamical parameters, i.e. parameters that you can edit during runtime.




### 3.3.1.4. DRC Log

The *DRC Log* window displays the results of the last Design Rules Check. A click on one of the module names listed under a warning/error message will bring the respective design window into focus (in the design panel), with the module selected and highlighted.

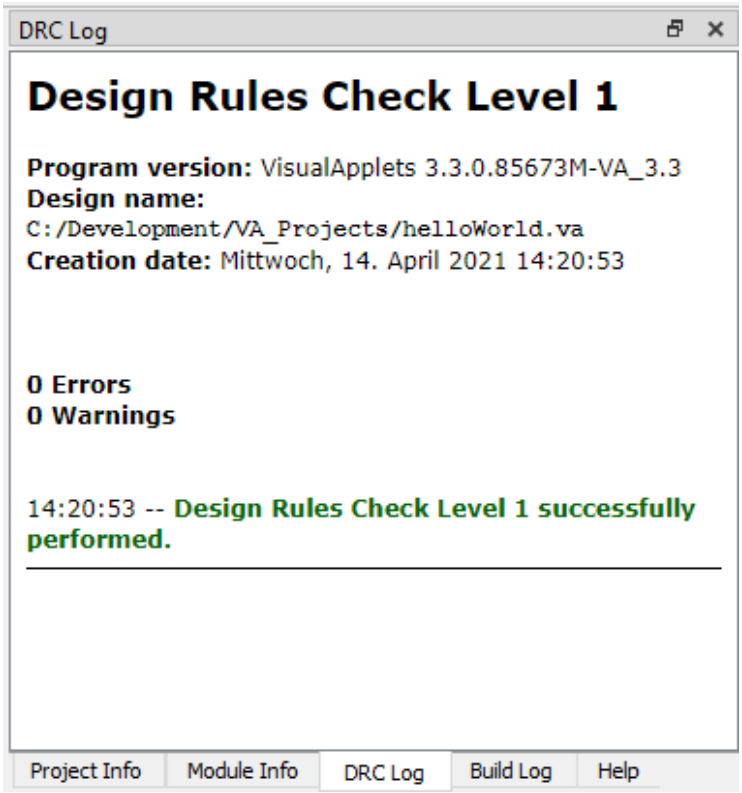



Figure 3.9. DRC Log Information

### 3.3.1.5. Build Log

The *Build Log* window displays the results of the last Build.

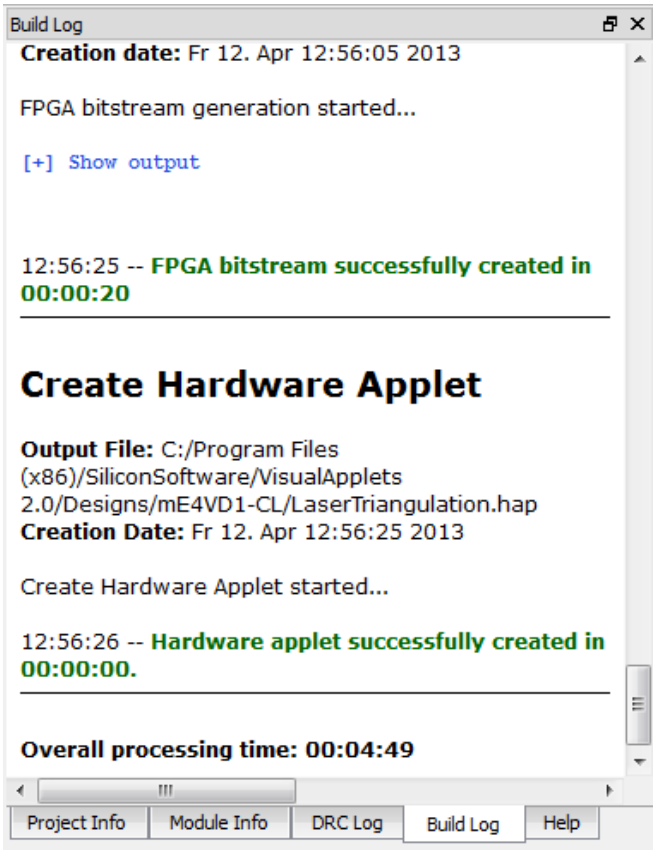



Figure 3.10. Build Log Information

### 3.3.1.6. Help

The *Help* window offers context-sensitive help on individual operators.

To open the context-sensitive help you can either:

1. Select the operator you want information on directly in you design and press **F1** to open and display the corresponding information in the *Help* window.
2. Select a operator from the *Operator Library* window. The *Help* window will always show the help for the current selection.

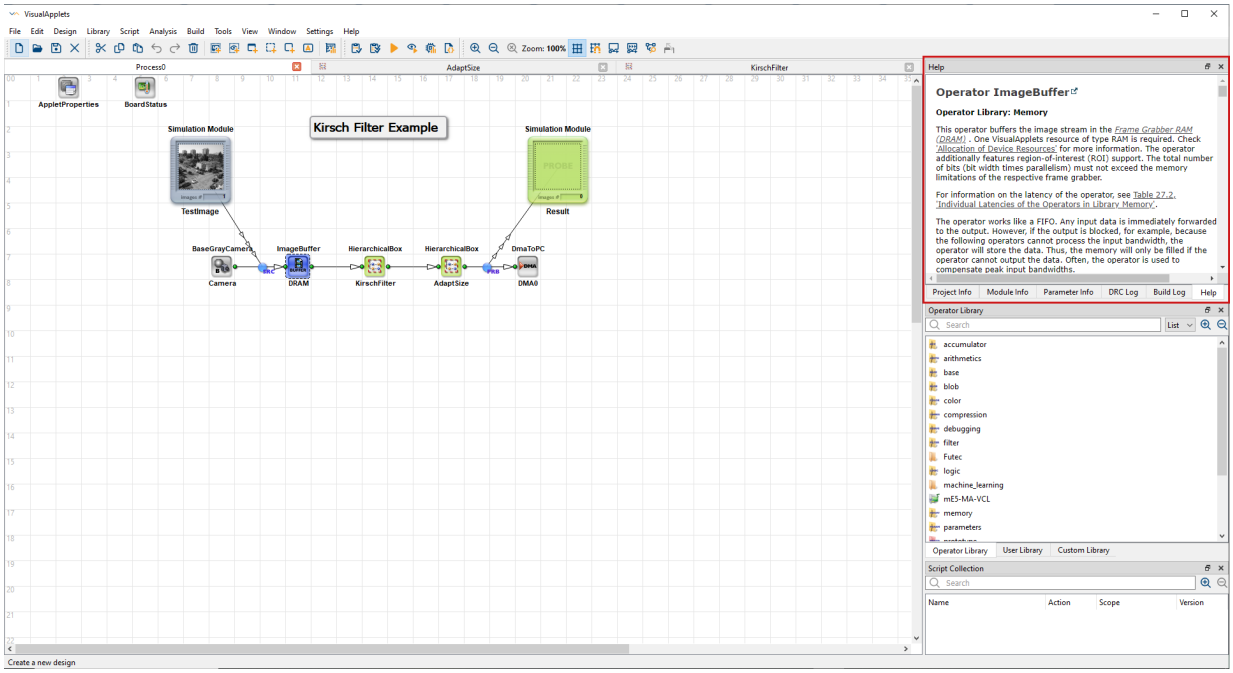



Figure 3.11. Example: Displaying Information on the MergeKernel Operator

If you press **F1** in any other window or click on a help button, the context sensitive user manual help for the current selected window or function will open. Of course, you can access the operator help directly form this user documentation, too. See Part III, 'Operator Reference'.

## 3.3.2. The Windows of the Library Panel

The library panel displays the Operator Library window, the User Library window and the Custom Library window in tabbed view.

All libraries (Operator Library, User Library, and Custom Library with all their sublibraries) are organized in a tree structure. Each library contains operators.

The visualization of the structure can be changed by selecting either "Tree", "List", or "Icon" view.

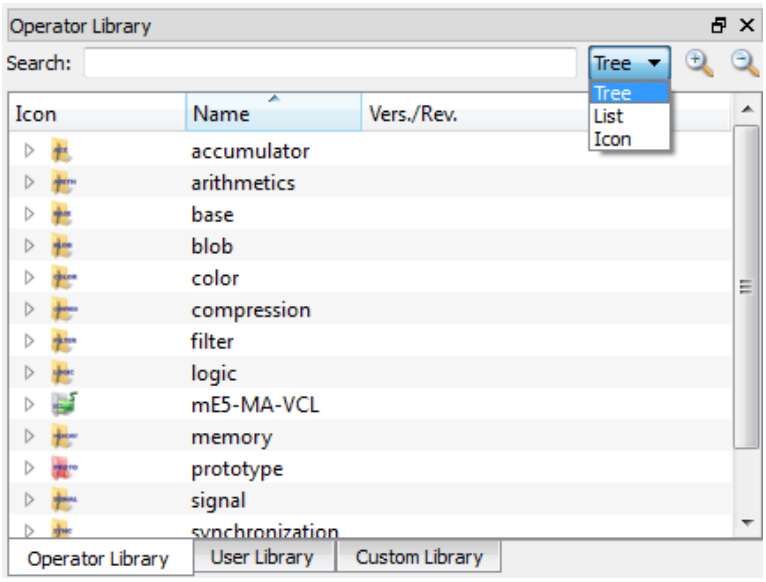



Figure 3.12. Library Panel with Operator Library on Display

In the search field, a search string can be entered to filter the list for only the required operators. **Camera** for example will list all operators containing **camera** in their name.

### 3.3.2.1. Operator Library

The Operator Library is a collection of sublibraries that contain VisualApplets operators, i.e. the individual image processing functions.

You can get detailed information on each operator by clicking on the operator name in the *Operator* Library window and clicking on the *Help* tab in the information panel.

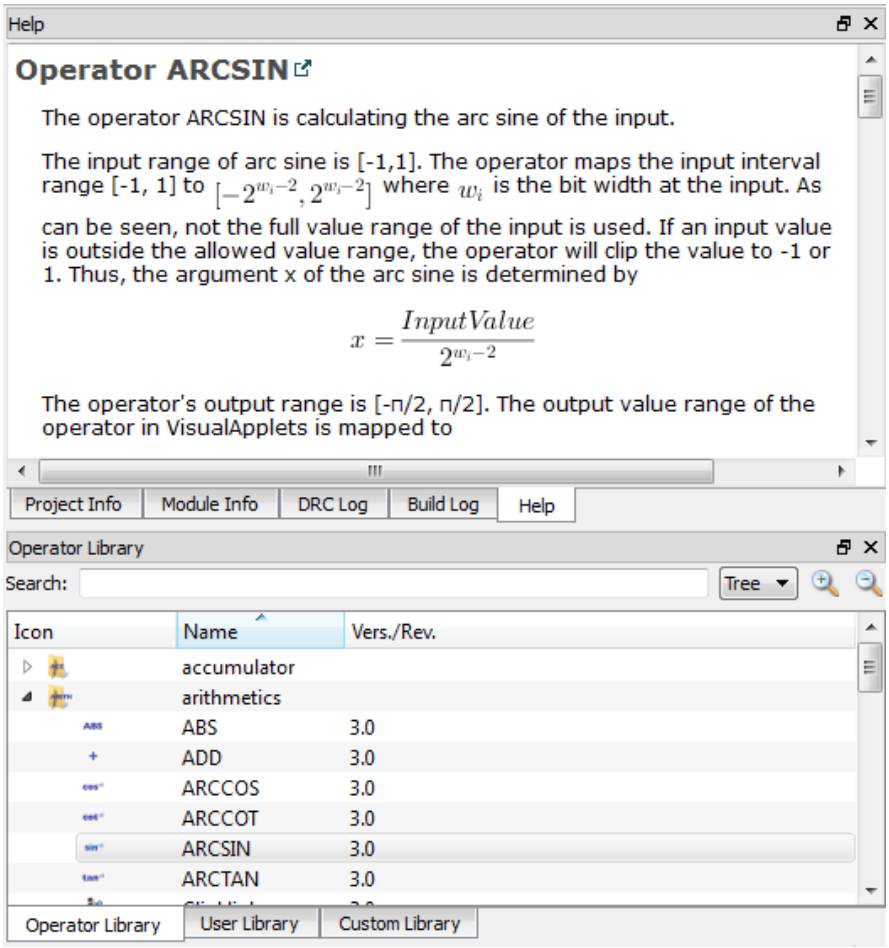



Figure 3.13. Library Panel with Operator Library on Display

See Section 3.4.4, 'Inserting Operators' on how to add new operators to a diagram. A detailed documentation of all operators is given in Part III, 'Operator Reference'.

### 3.3.2.2. User Library

The User Library can include custom libraries with user-defined combinations of operators (hierarchical boxes). For more information see Section 4.2, 'User Libraries'.

### 3.3.2.3. Custom Library

To import operators into the Custom Library you need either an **Expert** license or the **VisualApplets 4** license.

### 3.3.3. Adapting the Main Program Window

You have various options to adapt the main program window of VisualApplets to your personal preferences:

#### 3.3.3.1. Adapting Toolbars

Toolbars can be rearranged by drag and drop or added/removed by a right click.

#### 3.3.3.2. Restoring the Default Setting of the Main Program Window

If you accidentally arranged the windows or toolbars in a way you didn't want to, you can restore the default setting by **Window** -> **Reset Dock Windows** ( **Ctrl+O** ) or by clicking the corresponding icon in the View toolbar.

#### 3.3.3.3. Number of Files under File -> Recent Designs

Under **File** -> **Recent Designs** VisualApplets offers you a list of recently opened VisualApplets files (*.va). You can configure the actual number of files displayed.

To define how many recently opened files are displayed:

1. Go to menu **Settings** -> **System settings** and select category **Common**.
2. Under **Recent designs**, define the number of files you want to have displayed (up to 24).
3. Confirm with **Ok**.

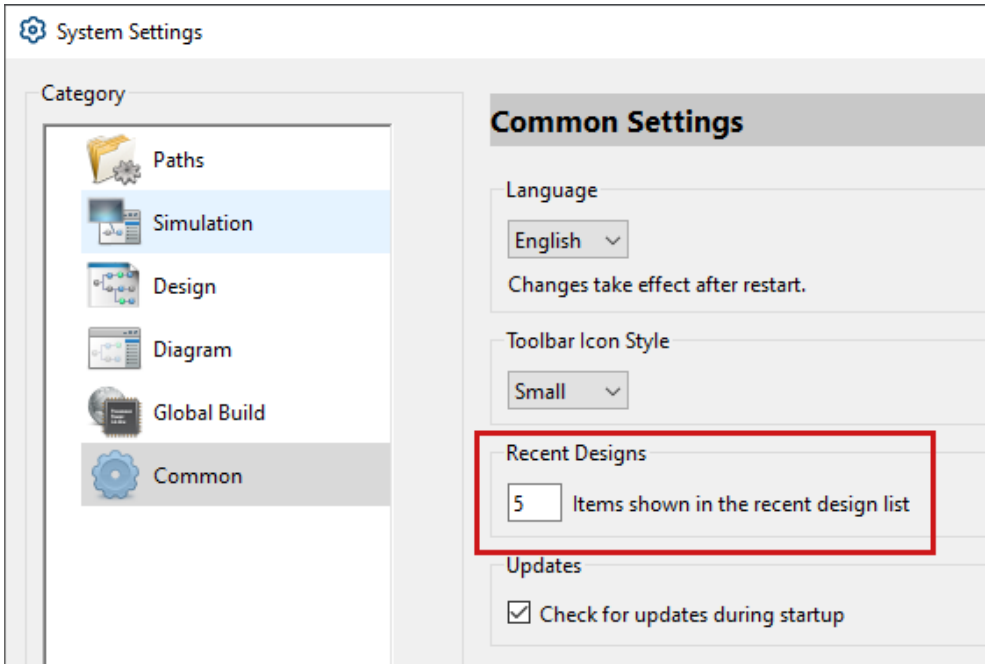



Figure 3.14. Configuring the number of displayed recent designs

## 3.4. Entering a Design

VisualApplets offers a graphical user interface for applet design. No scripting, no programming-code is required to be edited. The user interface is very simple and especially designed for easy learning by users. No technical background in programming is required.

### 3.4.1. Creating a New Project

To create a new VisualApplets project file:

1. Click **File** -> **New** ( **Ctrl+N** ) or use the icon **New** from the file icon bar in the main window. A *New Project* window will open which allows you to specify a project name.

2. Enter the project name. The project name will be the suggested file name later on.

3. You also have to specify the target hardware platform and the target runtime. If you are not sure which platform and runtime will be used, you can always change these settings later on ( **Design** -> **Change Platform** and **Design** -> **Change Target Runtime** ). Finalized applets can only be used in a runtime which corresponds to the target runtime defined in the applet.

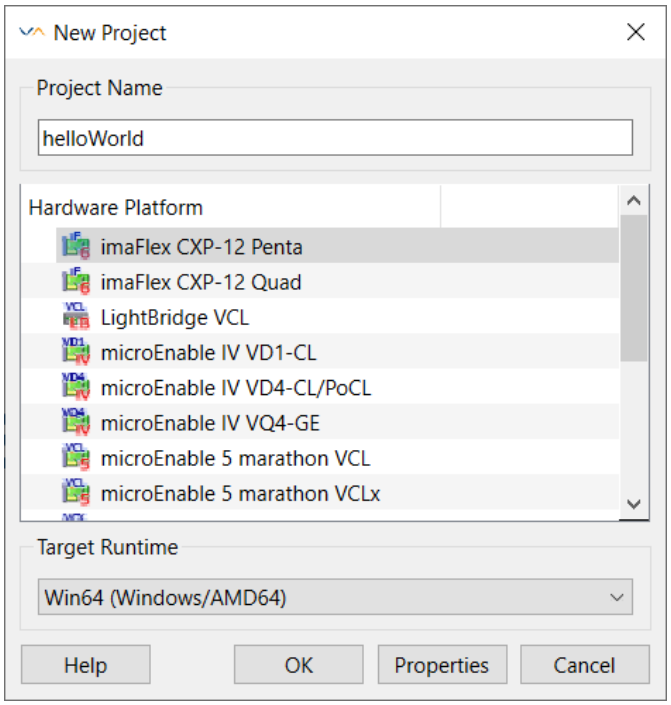



Figure 3.15. Start of a new Project

For each project, a version number (string) and a project description can be added.

4. Click **Properties** to open the *Project details* window. Here, you can enter the version of your design project and a project description. This information will be displayed in the *Project Info* window of the information panel.

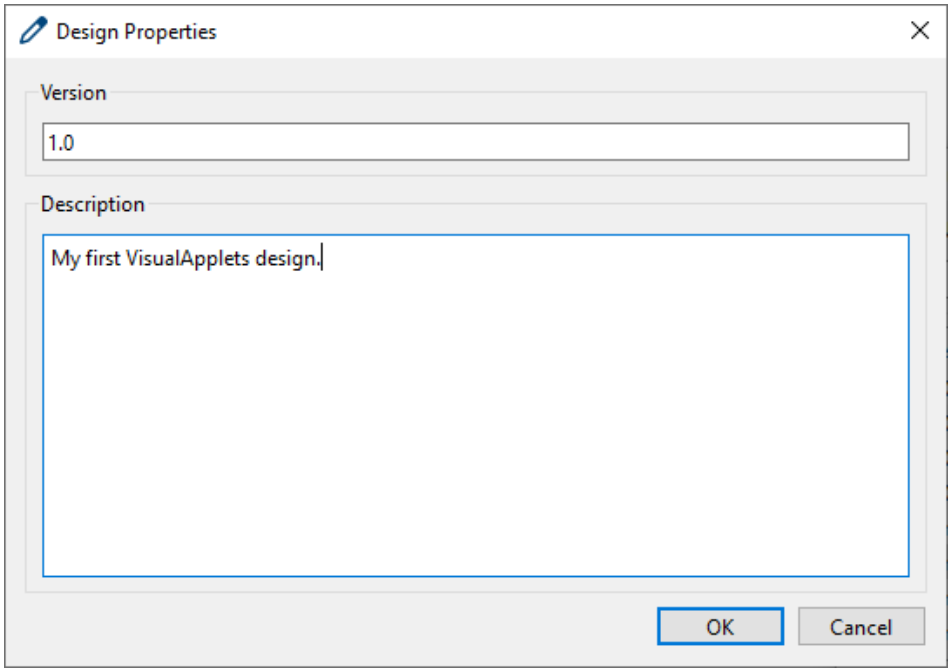



Figure 3.16. Edit Project Details

After all settings have been made:

5. Click **OK** to close all windows and to start the new project. VisualApplets will now open your new project.



**Tip**

Only one project can be loaded at a time. If you want to open several projects at the same time, simply use a second VisualApplets instance.

### 3.4.2. Opening an Existing Project

To open an existing project:

1. Click **File** -> **Open** ( **Ctrl+O** ) or use the icon **Open** from the file icon bar.
2. Select the VisualApplets project file from your file system an click **Open**.



**Opening Designs created in older VisualApplets versions**

If you are opening a project file which has been edited with a previous version of VisualApplets, the program might ask you to convert the file to the new version. For more information on migrating from previous versions, see Section 4.14, 'Migration from Older Versions'.

Alternatively, you can select a project you have currently worked on directly:

1. Go to **File** -> **Recent Designs**. VisualApplets offers you a list of recently opened VisualApplets files (*.va).
2. Select the project file you want to proceed working on.



**Number of displayed files**

You can configure the actual number of files displayed. How to do that, see section Section 3.3.3.3, 'Number of Files under File -> Recent Designs'.

### 3.4.3. Defining the FPGA Clock (mE5 marathon only)

If you are designing for another hardware platform than mE5 marathon, skip this section and proceed with the following one.

If you are designing for a mE5 marathon hardware platform, you can specify the FPGA clock frequency. You can select any value between 125 MHz and 312.5 MHz.

To define the FPGA clock:

1. In menu **Design**, select menu item **Change FPGA Clock**.

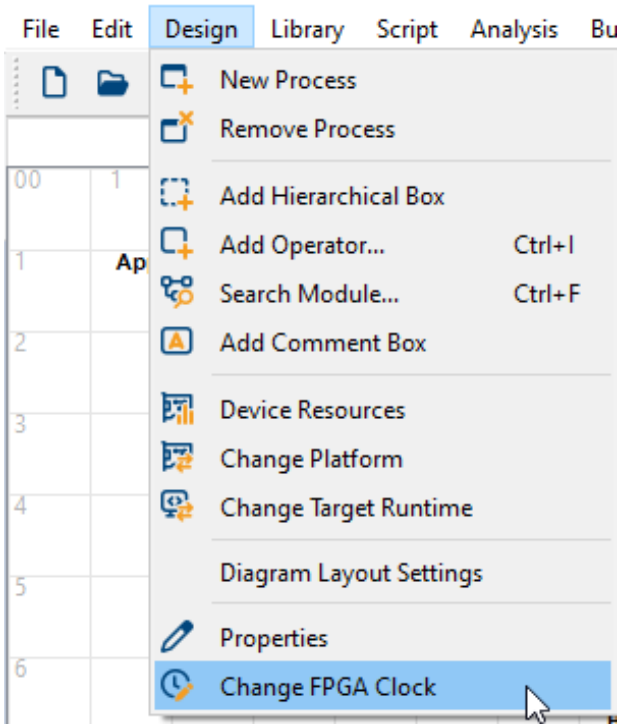



Figure 3.17. Menu Design, menu item Change FPGA Clock

2. Specify the desired frequency via slider or spin box.

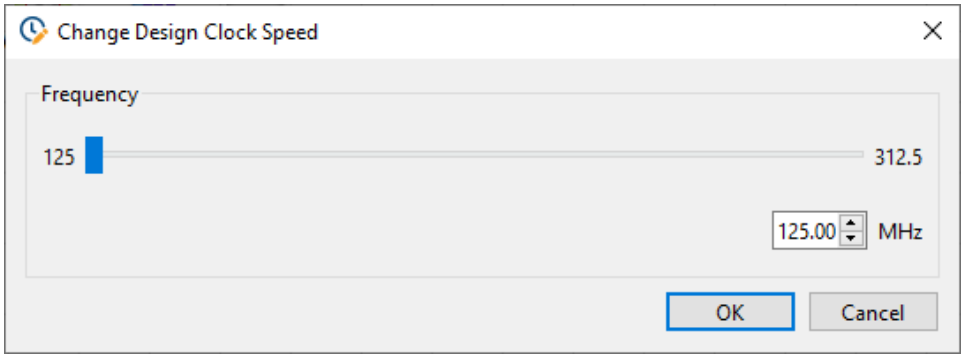



Figure 3.18. Slider and spin box for selecting FPGA clock frequency

The selected clock frequency (Design Clock) is displayed in operator *AppletProperties*.

### 3.4.4. Inserting Operators

Inserting new operators into a project is very easy:

1. Select the required operator in the library panel (see Section 3.3.2, 'The Windows of the Library Panel' if you want to know how to find the operator you are looking for).

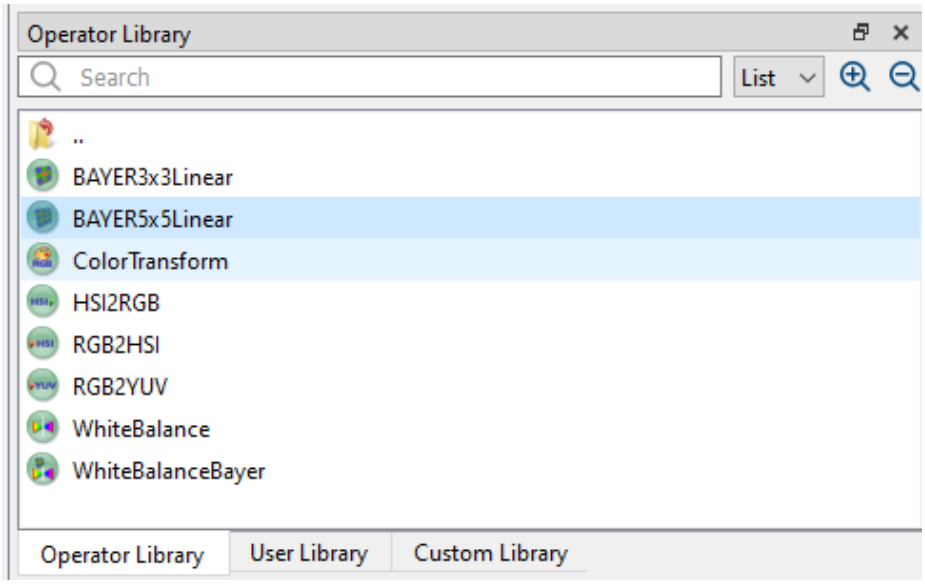



Figure 3.19. Operator Libraries

2. To add an operator from the library list into your diagram window, simply drag and drop the operator into the design window. The selected operator is now added to your project.



**Operator Instances in Your Design**

As soon as you drag and drop an operator into your design, the operator gets instantiated and you work with an instance of the chosen operator. This instance in your design we call a design module. You can give an individual name to each module of your design (the **module name**). This way, you can even give an individual name to each of several instances of the same operator.

3. Move the modules in your design into the right position by using drag and drop.



**Quick Insertion of New Modules**

There is an alternative method to add operators into a project. You can use the *Add Operator* window which you open via **Design** -> **Add Operator ...** ( **Ctrl+I** ). This window is especially useful if you need to add a series of operators you know by name. The shortcut for accessing the window, in combination with the window's search field facilitate the efficient and fast insertion of multiple operators. The operator you chose in this window is added to the design window currently in focus.

### 3.4.5. Deleting Modules

If you want to delete a module:

1. Select the module and press the DEL key, or right click the module and select **Delete** from the menu.

### 3.4.6. Adding Links

The modules of a project have to be connected via links. To connect your modules using links:

1. Click the output port of a module and hold the mouse button.
2. Move the mouse to the input port of the next module and release the mouse button.

Now, the link will be established. You can also connect modules from an output port to an input port, i.e. draw in reverse direction.

There are two alternative ways to connect modules which you might find efficient:

- Click the output port of the first module you want to connect. Then, click the input port of the module you want to connect to the first module. Now, the link will be established. Or,
- Drag one of the modules and drop it on the port of the module you want to connect it with. This will generate a link between the touching ports.



**Some Linking Rules**

- It is not possible to add more than one link to a single port.
- An input port always has to be connected to an output port. It is not possible to connect two input ports or two output ports.
- Loops are generally not allowed in projects. See Section 3.6, 'Rules of Links' for more information.

A drag and drop of an existing link to another port can change the connection. The link path of an established link can be changed by moving the line using drag and drop. However, new edges can't be inserted.



**Tip**

You can delete a link by selecting it and pressing the DEL key (or, right-clicking it and selecting Delete).

### 3.4.7. Clipboard

Modules and links in a project can be copied and pasted to and from the clipboard.

1. Select the operators and links to be copied, for example by drawing a rectangular selection area.
2. Copy the selection via **Edit** -> **Copy** ( **Ctrl+C** ) or from the context menu, select Copy.
3. To insert, select **Edit** -> **Copy** ( **Ctrl+V** ) or from the context menu, select Paste.

If you use the context menu, the inserted modules and links will be positioned at the current mouse position. Otherwise, the inserted modules and links are positioned in the center of the currently visible part of the design window currently in focus.

Modules are copied with all their parameter settings. Links between modules are copied, too. However, a link which is not connected to two copied modules can't be copied as it is not allowed to have single-sidedly connected links. **This can result in a changed output link format of a copied module.** Verify the parametrizations after using the clipboard.



**Copying & Pasting to Other Projects**

You can copy parts of a project to other projects in different program instances. However, you should not use the clipboard between different versions of VisualApplets.

Some operators (e.g., EventToHost) are available only for specific platforms. If you copy such an operator to a design for a target hardware platform that does not support this operator, DRC1 delivers an according error message:

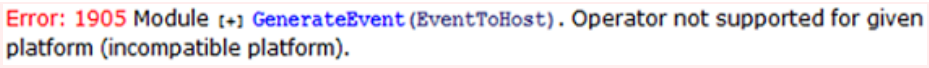



Figure 3.20. Error message in case an operator is not applicable for another hardware platform

### 3.4.8. Undo / Redo

Undo and redo operations are possible in VisualApplets.

If you want to undo your last action:

- Select **Edit** -> **Undo** ( **Ctrl+Z** ).

If you want to redo your last action:

- Select **Edit** -> **Redo** ( **Ctrl+Shift+Z** ).

The operations require some seconds to be processed.

### 3.4.9. Saving a Project

The project is **not** automatically saved.

1. To save your design, use **File** -> **Save** ( **Ctrl+S** ) or **File** -> **Save As...** (or use the according menu options / toolbar buttons).

You can save your design either in the `*.vad` file format or in the `*.va` legacy file format. If you want to open your design in VisualApplet version 3.3.2 or older, save your design in the `*.va` legacy file format. Basler recommends to always use the `*.vad` file format, if possible.

Designs saved in the `*.va` legacy file format are backwards compatible, which means you can still open them in VisualApplets and you can still save your design in the `*.va` file format. However, if your design in the `*.va` legacy file format contains a protected user library element, the content of the protected element isn't saved along with the design. In this case, the user library that contains this element must be imported in VisualApplets when you want to open the `*.va` legacy file format.



**File Name and Project Name**

If you save your design for the first time, or if you use the *Save as...* option, the *Save*/*Save as...* dialog will offer the project name as the name for your design file.

### 3.4.10. Navigating Your Design

A design might consist of a large network of modules in several design windows. Therefore, it is very helpful to have some navigation tools which facilitate to keep track with the details of your design.

These are the most important ones:

- Changing design windows

  To change between the windows in a design, you can use **Window** -> **Diagram Windows** ( **Ctrl+F8** ).

- Finding a module in your design

  To find a specific module in your design, you have two possibilities:

  - You can use the search options of the Module Info window. Click the *Module Info* tab in the information panel, and enter your search string. You don't have to enter a full module or operator name. Entering a few characters is often enough. Make sure you use the right filter for your search: Use the *Name* filter if searching for an individual module name. Use the *Type* filter if searching for the name of the operator your looked-for module is an instance of.
  - Alternatively, you can also use the *Search Module* window which you open via **Design** -> **Search Module...** ( **Ctrl+F** ). Enter your search string. You don't have to enter a full module or operator name. Entering a few characters is often enough. Make sure you use the right filter for your search: Use the *Name* filter if searching for an individual module name. Use the *Type* filter if searching for the name of the operator your looked-for module is an instance of.

- Selecting a module:

  To select a module in a design window, click the module.

- Using the module menu:

  To use the available menu for a module, right-click the module and choose the option you need from the menu.

## 3.5. Data Flow

In Section 3.1, 'Basic Principles' the VisualApplets idea of building FPGA applications using operators and links was explained. This chapter will explain the data flow model of FPGA data processing.

FPGA implementations differ from software programs run on a microprocessor. All implemented functionalities of an FPGA implementation exist and can be used at the same time (i.e., in parallel) while the functions of a microprocessor program are executed sequentially. This is a very important difference and one of the tremendous advantages of FPGA implementations. As everything is working in parallel, data is processed in a pipeline structure. As mentioned, modules (instantiated operators) are connected by links. Each module starts the processing of image or signal data as soon as data is available. The results are forwarded to the next module via link. Most modules do not buffer the input image data. They output the calculated results as soon as the information is available. Let's have a look on our project from the 2. *Getting Started*guide once again:

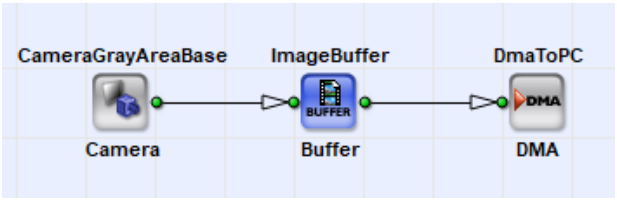



Figure 3.21. Simple VisualApplets Design

The project consists of three modules; namely, the camera, a buffer, and a DMA operator. The camera operator is an image source module. It receives the images sent from the camera. But instead of collecting a full frame and outputting the image after full acquisition, the module will forward the pixels to the next module as soon as they arrive. The buffer module *ImageBuffer* is a buffer which can store image data but will output available data as soon as the output is not blocked. The DMA module transfers the images to the host PC.

The result of this pipeline structure is that image pixels are transferred to the host PC while other pixels of the same image are still transferred from the camera. Thus, no images are stored inside the modules. The advantage of this non-buffered pipeline is that all modules can run in parallel and are efficiently used. Furthermore, the latency (the time a pixel needs until it is fully processed) is reduced to a minimum.



**Note**

Always keep in mind that all modules can run in parallel and output their results as soon as they are available and the output is not blocked. Imagine the pipeline like a water pipeline with valves, branches, and small and wide tubes.

In difference to a microprocessor program the number of operations, i.e., the number of operators, will not influence the processing speed as everything is running in parallel. However, the more operators are used in an applet, the more hardware resources are required.

### 3.5.1. Bandwidth of an Applet

The bandwidth of a pipeline in an applet depends on the processing speed of the operators and on the connecting links. A link has several “link properties” (see Section 3.7.2, 'Link Properties'). One of them is the parallelism which defines the bandwidth. The parallelism defines how many pixels are transferred in parallel between two operators in one design clock cycle. The higher the parallelism, the higher the bandwidth. Operators are automatically adapted to meet the required bandwidth of a link; if this is not possible, they redefine the bandwidth.

Let's assume a simple example. A parallelism of four will result in four pixels being transferred in parallel. This means the first four consecutive pixels of an image are transferred from the camera module to its successive module (in our case the buffer) in parallel. Next, the next four pixels are transferred from the camera module to the buffer module. Meanwhile, the first four pixels have been processed in the buffer module and are forwarded to the next module in the image processing chain.

As mentioned before, the parallelism defines the number of pixels transferred in parallel in one clock cycle of the design clock frequency. The frequency depends on the used hardware device. For example, the frame grabbers of the microEnable IV series use a design clock frequency of 62.5MHz.



**Calculating the Bandwidth**

The bandwidth b of a link is determined by the product of the parallelism p and the frequency f:

$$b = f \times p$$

A parallelism of four will therefore result in a bandwidth of $62.5MHz * 4Pixel = 250MPixel/s$ . If we assume a bit width of 8 bit per pixel, this will result in a bandwidth of 250MB/s. A list of the basic design clock frequencies for all hardware platforms can be found in Appendix A, '*Device Resources*'.

Operators may change the parallelism between the input and the output link. Thus, the bandwidth is not constant throughout the design. That's because the required bandwidth might change. Suppose an operator reduces the image size. Thus, the required output bandwidth is reduced, too.

The bandwidth calculated with the formula above is a theoretic value. The actual bandwidth is slightly less than the theoretic value.



**Important**

Some operators cannot process the full bandwidth given at the input link. You will find detailed information for all operators concerned in the Part III, 'Operator Reference'.

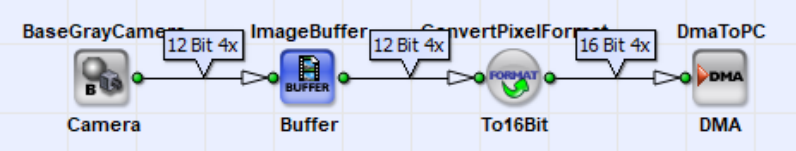
Note the difference between the bandwidth and the latency. The latency is defined individually by each operator and mostly depends on the algorithmic implementation.



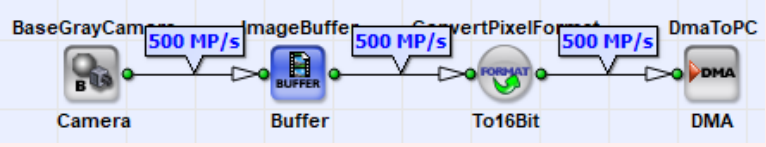
**Visualization on GUI**

For visualization of the according link properties, VisualApplets provides two GUI buttons in the toolbar of the program window:

**Display Link Info** displays the bit width and the parallelism for every link in the diagram:




**Display Link Throughput** displays the maximum pixel throughput in megapixels per second for every link in the diagram:




## 3.5.2. Pixel Order

As previously explained, in VisualApplets, pixels are transferred through the pipeline one after another. If a link parallelism is greater than one, multiple pixels are transferred in parallel. The order of the

transfer of pixels in images or lines is the following: In general, pixels of frames (two-dimensional images) are transferred from a camera starting with the first pixel at the upper left corner, and finishing with the last pixel at the bottom right corner. If VisualApplets operators require the pixel position for their processing, the same order, that is, left -> right and top -> down, is expected. The following figure illustrates this order:

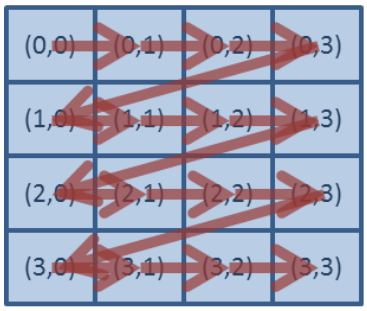



Figure 3.22. Pixel Order

However, some sources like cameras do not comply with this order. In these cases, VisualApplets operators and designs can be used to correct the pixel order. To do this, you should have some knowledge on the protocol of a pixel transfer. You get the according information in the section Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure'.

### 3.5.3. Image Protocols, Image Dimensions and Data Structure

The **Image Protocol** defines the image dimension and data structure of the transfered pixel and data. The **Image Protocol** is another link property besides the previously mentioned link property **Parallelism**.

There are three types of image protocols:

- "2D" **`VALT_IMAGE2D`**

  The 2D image protocol is used for the transfer of images, mostly used with area scan cameras. A link transports the information

  - when a pixel is transferred (pixel valid signal)
  - when a line is completed (end-of-line signal)
  - when a frame is completed (end-of-frame signal)

  Thus, a 2D image can have:

  - an arbitrary number of pixels in a line
  - an arbitrary image height

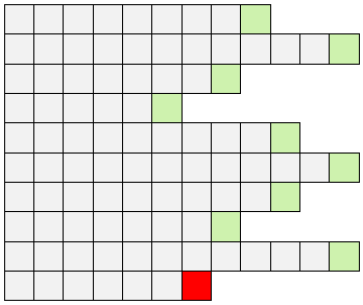


Figure 3.23. Model of a 2D Image Protocol

The pixel position itself within a line or row is not transfered. The position results out of the number of antecedent lines and antecendent pixels within the current line. It is not possible to have gaps between the pixels in a line. It is possible to have arbitrary line lengths. A line may have no pixels, i.e., empty lines are possible. As a minimum, a frame has to consist of one empty line.

- "1D" **VALT_LINE1D**

  The 1D image protocol is used to transfer lines, mostly used with line scan cameras. A link transports the information

  - when a pixel is transfered (pixel valid signal)
  - when a line is completed (end-of-line signal)

  Thus, a 1D image can have:

  - an arbitrary number of pixels in a line
  - an unlimited image height

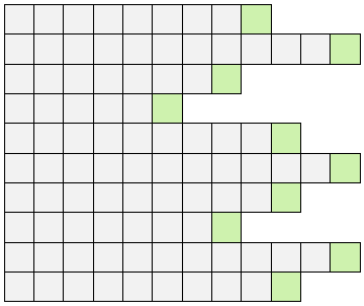


Figure 3.24. Model of a 1D Image Protocol

Again, the pixel position itself within a line is not transfered. The position results out of the number of antecedent lines and antecendent pixels within the current line. It is not possible to have gaps between the pixels in a line. It is possible to have arbitrary line lengths. A line may have no pixels, i.e., empty lines are possible.

- "0D" **VALT_PIXEL0D**

  In the 0D image protocol, no information on image dimensions is preserved. It is simply a stream of pixels. Still, time-gaps between pixels may exist, i.e., a pixel comes with a valid signal. The 0D protocol is mostly used for data transfers such as measurement results.



Figure 3.25. Model of an 0D Image Protocol

- "Signal" **VALT_SIGNAL**

  In the signal protocol, transfers are reduced to single bit data transfers which are always valid (valid at every clock cycle). Thus, the protocol does not include any control signals such as **pixel valid** or **line/frame completed**. The signal protocol is used for signal operators used in trigger and signal processing systems.

The image protocol is a link property. Each operator decides individually which link properties are accepted at its inputs and available at its outputs. See Section 3.7.2, 'Link Properties' for more information about link parameterization.

### 3.5.4. Flow Control

In the previous sections, the pipeline structure of VisualApplets designs was explained. As mentioned before, data is transferred between the modules of a project via links. As soon as a module has

processed an input pixel and is finished with calculating the output value, the result is output at the output link(s). However, the next module might not be able to process the data as it is still processing another pixel. In this case, the flow control of VisualApplets is applied and the pipeline is blocked. Thus, modules can block their inputs. In this case the antecedent module will not output its results and will propagate the blocking state backwards in the pipeline.

Let's have a look at the simple example shown in Figure 3.21, 'Simple VisualApplets Design'. Suppose a slow PC is used which cannot process the bandwidth generated by the camera. In this case, the *DmaToPC* module will not be able to transfer the data to the host PC. Thus, it will block its input from time to time. The blocking signal is propagated in the design up to the image buffer module. Now, the image buffer will not output any data while the blocking is active. As the *ImageBuffer* module is a buffer, all further incoming data will be buffered and the fill level of the buffer will increase. When the camera stops sending data, no new input data is transfered to the buffer and data will be output until it is empty.



**Tip**

Again, you can imagine the flow control like a pipeline of water. If a valve is closed, no more water can be transported. A buffer operator is like a reservoir which is filled with water if the drain cannot consume the input stream.

## 3.6. Rules of Links

Any diagram created in VisualApplets is a network of operator instances (modules). It may combine several sub-networks.

Like other programming languages, the diagrams you create in VisualApplets are subject to a set of rules with allowed, not allowed, and limited constructs. The following sections define the basic rules of connecting modules.

### 3.6.1. Operator Types

At first, let us have a look at the different types of operators you are working with.

In VisualApplets, you have three **types** of operators:

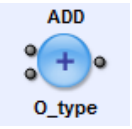
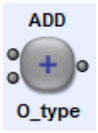
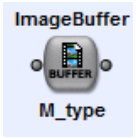
| | | | |
|---|---|---|---|
| The universal type "O" (round shape): |  | |  |
| The more complex type "M" (square shape): |  | In older versions of VisualApplets and some examples in documentation oval shape: |  |
| The special type "P" (square shape): |  | In older versions of VisualApplets and some examples in documentation oval shape: |  |

Table 3.1. Operator Types

You can see by the shape of an operator if it belongs to the universal "O" type or to one of the more complex "M" or "P" types. (For information on the type of a specific operator, check the documentation for that operator in Part III, 'Operator Reference'.)

The functionality of O-type operators is rather simple. Therefore,

- O-type operators do not change the number of pixels/data between input and output. The number of input pixels/data of all input links is equal to the number of output pixels/data, i.e. for each input value, the operator will output a value.
- O-type operators do not add a delay between input and output.
- O-type operators can be linked as desired in series or in parallel.

The functionality of M-type operators is more complex:

- They can change the number of pixels/data between input and output.
- They may delay the output.
- Linking of M-type operators requires attention to the rules of links and to the functionality of the linked operators.
- M-type operators may block the input.
- Every source operator in a diagram (such as e.g. CameraGrayAreaBase) is an M-type operator.
- Every final-destination operator in a diagram (such as e.g. DmaToPC) is an M-type operator.

P-type operators are similar to M-type operators, but

- P-type operators do not reduce the bandwidth. (For more information on this issue, see Section 3.6.9, 'Infinite Sources / Connecting Cameras'.)
- P-type operators will not generate more data than given at the input(s).
- P-type operators never block the input actively. (See Section 3.6.8, 'P-Type Operators' for detailed information on the blocking behavior of P-type operators.)

In the following, all rules defined for M-type operators also apply for P-type operators if no exception for P-type operators is mentioned.

### 3.6.2. O-Type Networks

An O-type network is a combination of O-type modules. In an O-type network, modules may be combined in series as well as in parallel. **An O-type network always starts and ends with M-type or P-type modules.** On the basis of this definition, an important rule can be defined:



**O-Type Network Rule**

In an O-type Network, no matter how complex it might be, each O-type module has to be sourced by the same M-type or P-type module. It may be connected to this source via other O-type modules or even O-type (sub)networks. (Please note that this rule does not apply to signal links.)

The following figure shows an O-type network including parallel and serial connection of modules. As you see, all O-type modules are sourced by the same M-type module (*M_source*).

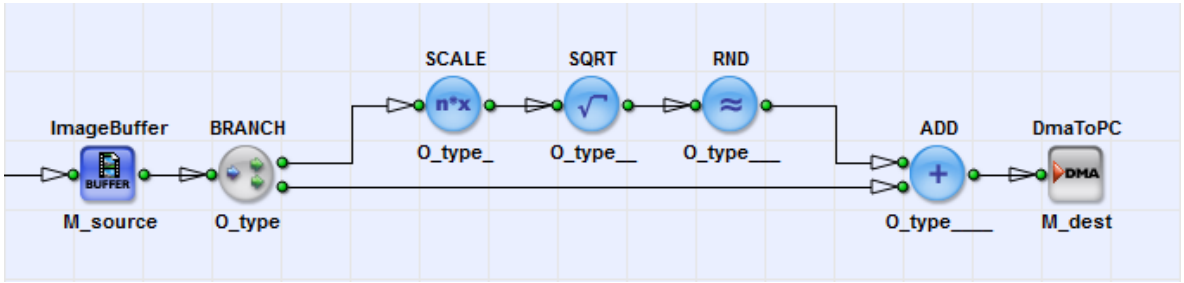



Figure 3.26. O-Type Network

In contrast, the next figure shows an O-type network which violates the O-Type Network Rule as the *ADD* module is sourced by different M-type modules (*M_source_0* and *M_source_1*).

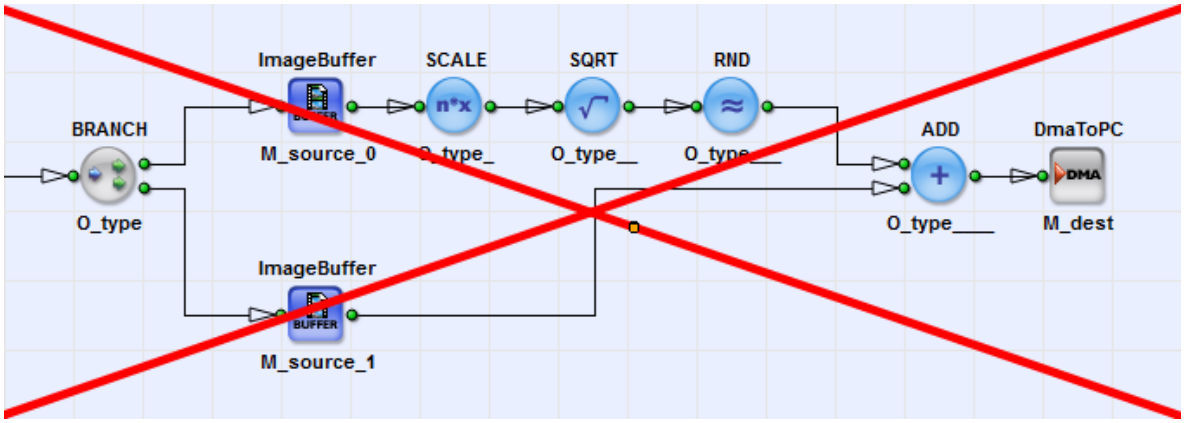



Figure 3.27. Failing O-Type Network

VisualApplets (version 2.2 and higher) reliably indicates directly in the design diagram whenever a synchronization rule is violated. The respective links are highlighted in purple. The links are highlighted from the operator that receives not correctly synchronized data back to the according M operator sources.

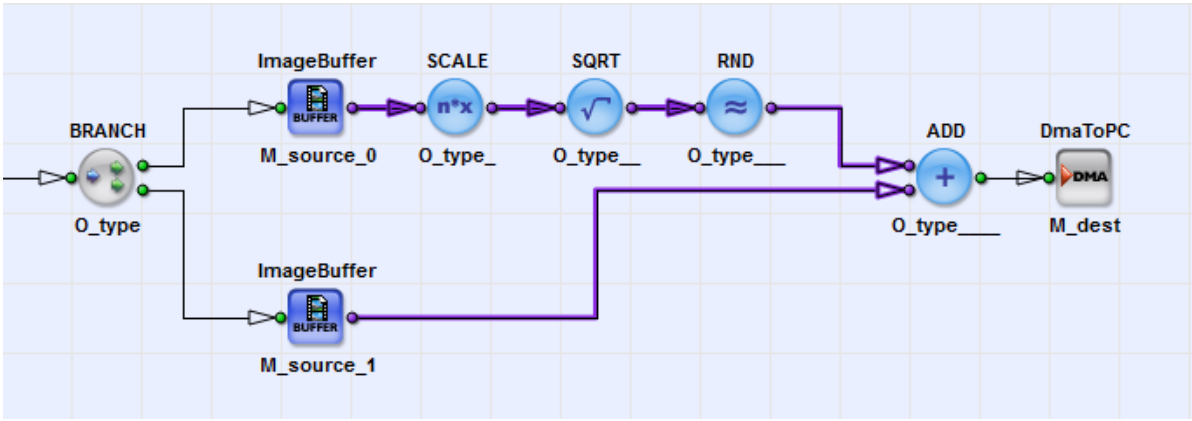



Figure 3.28. Display of not correctly synchronized data flow in VisualApplets 2.2 and higher

DRC 1 and DRC 2 display according warnings.

For example, for the above design, DRC 1 creates the following warning:

```
Warning: 1310 Module [+] O_type____(ADD) node: I001 is not properly synchronized.
```

**Exception: Signal Links**

The O-type network rule does not apply for O-type operators transporting signal links (see Section 3.6.10, 'Differing Rules for Signal Links').

### 3.6.3. M-Type Networks

Any network of modules that contains M-type modules not only as source and destination, but also in between for image processing purposes, is called an M-type network. In M-type networks, M-type modules can be linked in parallel as well as in series. M-type modules can also be branched without limitations. M-type networks may turn out pretty complex, as parallel linking of M-type modules may demand synchronization of image dimensions and synchronization of timing. (Remember: M-type modules can change the number of pixels/data between input and output; they also may delay the output.)

The following figure shows a very simple example of an M-type network. The M-type modules of this diagram are only connected in series, or branched. Since the only O-type network in the diagram is sourced by the same M-type module (*ImageBuffer*), the diagram follows all diagram rules we know so far:

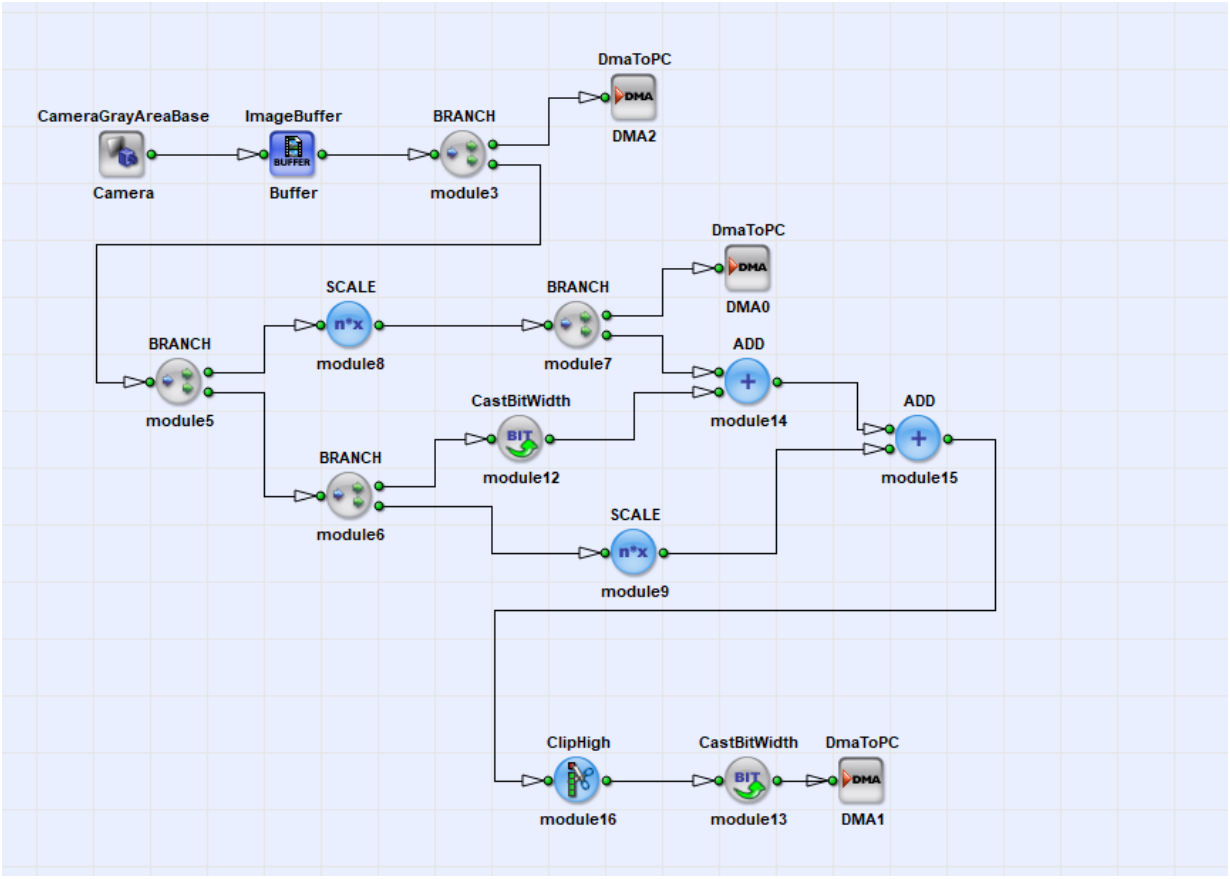



Figure 3.29. M-type and O-type Network

In most cases, M-type networks are much more complex. If, in contrast to the example above, M-type modules are linked in parallel (i.e., the merging paths are sourced by different M-type sources), we have to synchronize the timing and the image dimensions. This is done by M-type operators using multiple inputs.

## 3.6.4. M-type Operators with Multiple Inputs

If you want to merge networks sourced by different M-type sources, you have to use M-type operators with multiple inputs. (You can't use O-type operators with multiple inputs as this would violate the rules for O-type networks as described above.) But before going into detail on synchronization, let's have a look at M-type operators with multiple inputs in general.

M-type operators with multiple inputs can have two kinds of inputs:

- Synchronous inputs
- Asynchronous inputs

**Synchronous Inputs and Synchronous-Input Groups**

If some inputs of a module are synchronous, these synchronous inputs form together one synchronous-input group. Thus, an synchronous-input group is a set of synchronous inputs. The inputs of an synchronous-input group always have to be sourced by the same M-type source through an arbitrary network of O-type operators. An M-type operator can have an arbitrary number of synchronous-input groups.

On the picture below, you see an example of the use of an M-type operator with a synchronous-input group. Both inputs of the *RemoveImage* operator are sourced by the same M-type operator (ImageBuffer). The two inputs together form the group of synchronous inputs. The operator *RemoveImage* does not allow to use different M-type sources at its input ports.

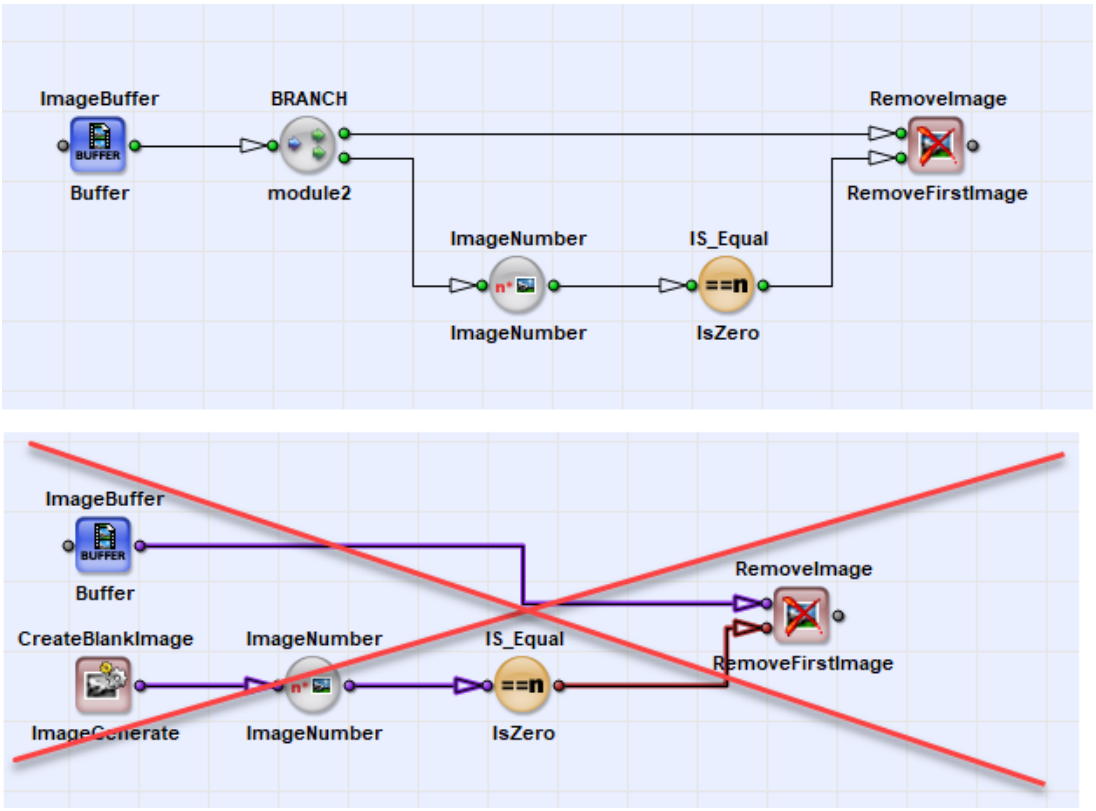



Figure 3.30. M-type Operator with One Synchronous Input Group

### Asynchronous Inputs

Asynchronous inputs are not synchronous. They may be sourced by different M-type operators through an arbitrary network of O-type operators. Moreover, some M-type operators with multiple inputs actually have to be sourced by different M-type operators. Whether an M-type operator with multiple inputs has to be sourced by different M-type operators or not depends on the features of the operator itself. You find detailed information on all operators in Part III, 'Operator Reference'. An M-type operator can have an arbitrary number of asynchronous inputs.

The following screenshot shows a typical example of an M-type operator with two asynchronous inputs. Both inputs of the *SYNC* module are sourced by different M-type sources, namely *DRAM0* and *DRAM1*.

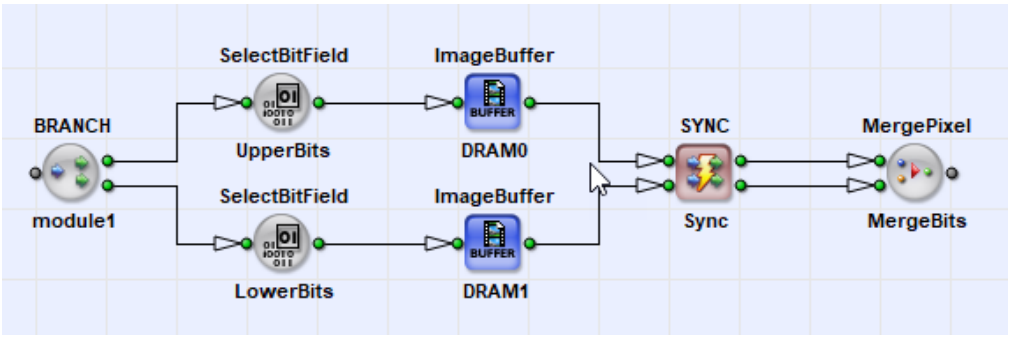



Figure 3.31. M-type Operator with Asynchronous Inputs

The SYNC module is followed by a *MergePixel* operator. This operator can't be used to merge the two M-type sources without the SYNC operator in-between.

**M-Type Operator Inputs**

The inputs of M-type modules may be sourced by different M-type sources.

However, the inputs of a synchronous-input group have to be sourced by the same M-type source through an arbitrary network of O-type operators.

The documentation for each M-type operator with multiple inputs explains the possible or required inputs for the operator in great detail. For a comprehensive description of all operators, see Part III, 'Operator Reference'.

### 3.6.5. Synchronization of Different Image Dimensions

M-type operators may synchronize different image dimensions at their inputs. One important example is the operator *SYNC*. See Part III, 'Operator Reference' for a detailed description of this operator.

### 3.6.6. Timing Synchronization

For O-type networks, VisualApplets automatically performs timing synchronizations.

For M-type networks, the synchronization has to be carried out by M-type operators.

If a timing synchronization is not designed correctly,

- a so called deadlock-condition will occur,
- the bandwidth is not sufficient, or
- data will get lost.

**Ensure correct timing synchronization**

Since the timing synchronization depends on the behavior of the algorithm you implement in VisualApplets, you have to make sure the timing synchronization in your diagram is correct.

In the following examples, we will have a closer look at timing synchronization.

1. **Synchronization of Independent Sources**

   Let's have a look at a simple two-camera stitching example which is shown in Figure 3.32, 'Synchronization of Independent Sources'. In the example, two cameras are used as a source. The *ImageBuffer* operators will buffer the data. Finally, the M-type operator *InsertLine* will multiplex the lines of both cameras into one output link. This operator merges the two parallel links.

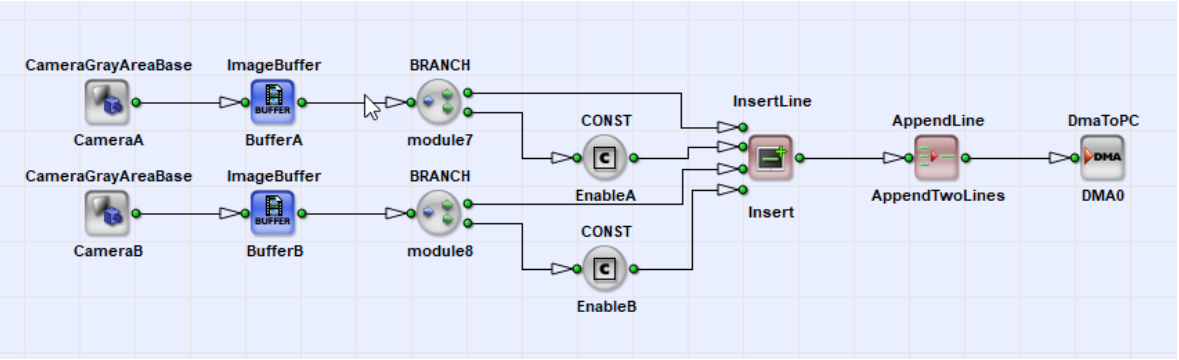



Figure 3.32. Synchronization of Independent Sources

Imagine what happens if the two cameras are not synchronous. Suppose the frame rate of one camera is higher than the framerate of the other. The *InsertLine* operator will strictly multiplex the

lines, i.e. will use the inputs one after another. Due to the different frame rates, one of the DRAM buffers will need to actually buffer data. Its fill level will be constantly increasing. As soon as an overflow condition occurs, no more data can be buffered, and data will get lost.

> **Ensure Equal Data Rate**
>
> Thus, if asynchronous streams are merged together, ensure that the line rate/frame rate/pixel rate is equal and can be synchronized.

2. **Synchronization of Data from the Same Source**

Some M-type operators create a delay until they can output their results. A prominent example is the operator *FIRkernelNxM* which is used in the diagram we explore in the following figures. This operator outputs a pixel together with its surrounding neighbors to form a kernel. To output the neighbored pixels, the operator has to wait until they (the required neighbored pixels) are available at the input. It will store the intermediate results. In our example, the FIRkernelNxM operator can output the first results only after four full lines have been fed into the operator. Thus, the operator generates a delay of four image lines in its current configuration.

Let's have a closer look at this example.

### Precondition

The first pixels are send from a camera through the buffer into the branch. Here, data is duplicated and forwarded to the *SYNC* and *FIRkernelNxM* modules. As explained, the *FIRkernelNxM* has a delay of four lines and therefore is not able to output the first pixels immediately:

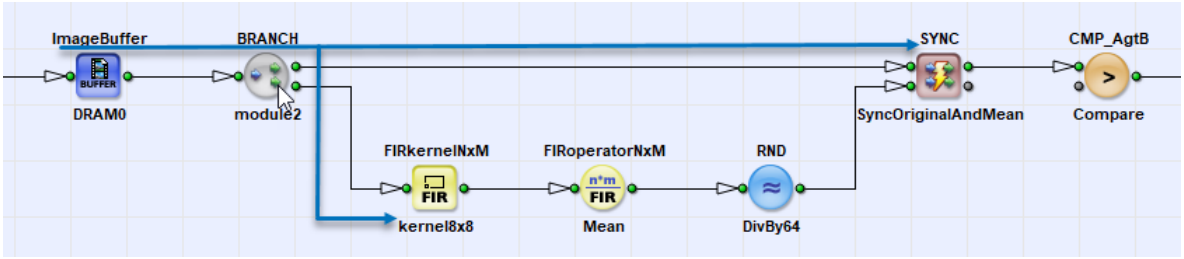



Figure 3.33. Deadlock at SYNC, figure a

### Deadlock

The SYNC operator tries to synchronize both inputs. As no data is coming from the second input, it generates a STOP-signal at the first input. This STOP-signal is propagated backwards in the pipeline up to the buffer. Now, we have the deadlock condition: The kernel operator requires more data until it is able to output data, but no new data are available because the SYNC blocks all further inputs.

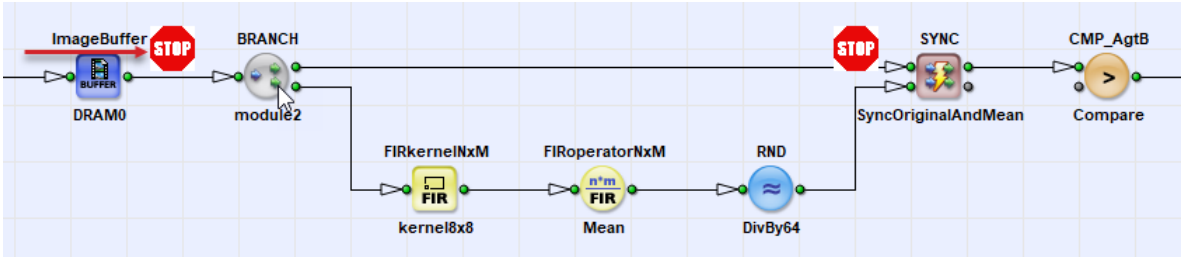



Figure 3.34. Deadlock at SYNC, figure b

### Fixing the Deadlock

To solve the deadlock, we have to add a buffer to the upper path, so that the delay of the kernel operator can be compensated as shown in the following:

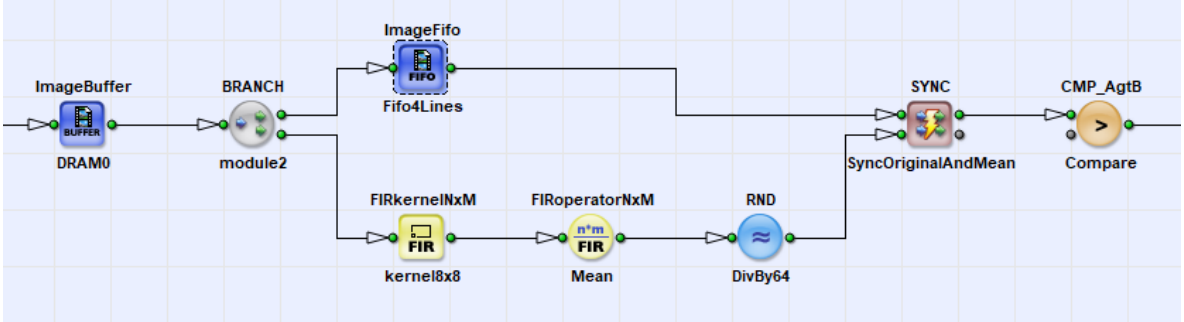



Figure 3.35. Fixed Deadlock

**Avoiding the Deadlock**

The best solution is to avoid synchronization at all. This is often possible. In the solution example below, there is only a serial pipeline of M-type operators in the diagram and therefore, no synchronization at the merge is required.

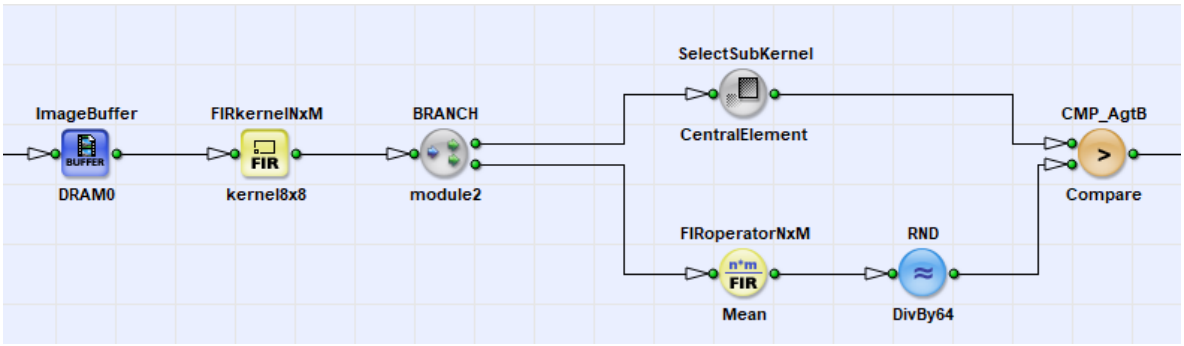



Figure 3.36. Deadlock Avoided

## 3.6.7. Bandwidth Bottlenecks

Another important phenomenon in M-type networks is the reduction of bandwidth in an operator together with a possible propagation of this limitation backwards in the network of modules. The following figure shows a diagram which will generate a bandwidth limitation.

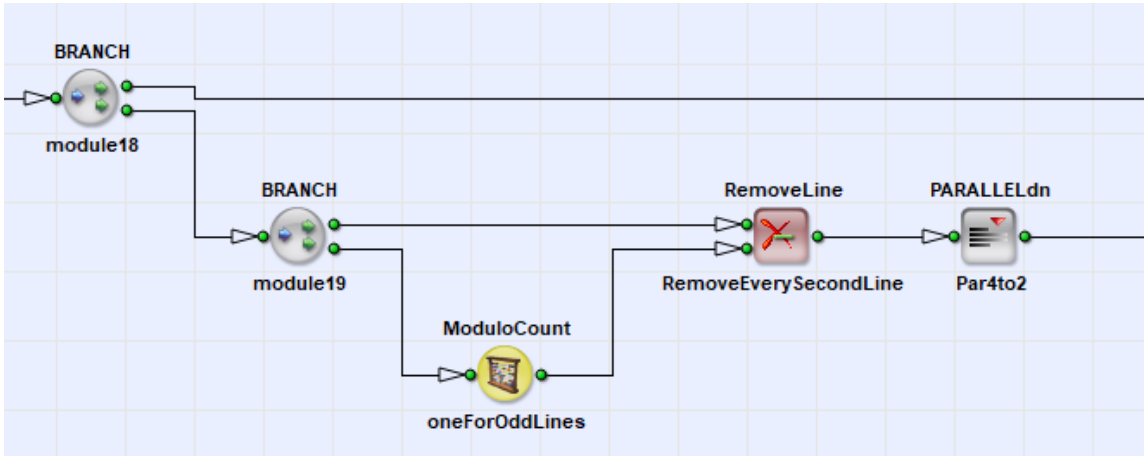



Figure 3.37. Bandwidth Limitation

In this diagram, the combination of *ModuloCount* and *RemoveLine* operator will delete every second line. As we know from Section 3.5.1, 'Bandwidth of an Applet', the maximum bandwidth depends on the used parallelism. After the *RemoveLine* operator, the required bandwidth has been reduced by a factor of two. Therefore, we can reduce the parallelism by a factor of two using operator *PARALLELdn* ("Parallel down"). However, this will also reduce the bandwidth in the upper path by factor two for every second line.

While a line is deleted at the *RemoveLine* operator, no pixels are fed into the *PARALLELdn* module. The full parallelism as given at the input of the diagram can be used in both paths. In contrast, when a line is not removed by the *RemoveLine* operator, the pixels of the current line are forwarded to the input of the *PARALLELdn*. *PARALLELdn* will now reduce the parallelism by two, i.e., it requires the double number of cycles to output a value compared to the input. Thus, the *PARALLELdn* operator will block its input every second clock cycle. As no buffer is used between the first *Branch* and the *PARALLELdn* operator, this blocking signal is propagated backwards in the pipeline. Therefore, the upper path is blocked every second clock cycle, too.

A solution to this problem is to add an extra FIFO before the *PARALLELdn* operator which is able to buffer a minimum of one line and thus compensates the peak and idle periods. The following figure shows this solution:

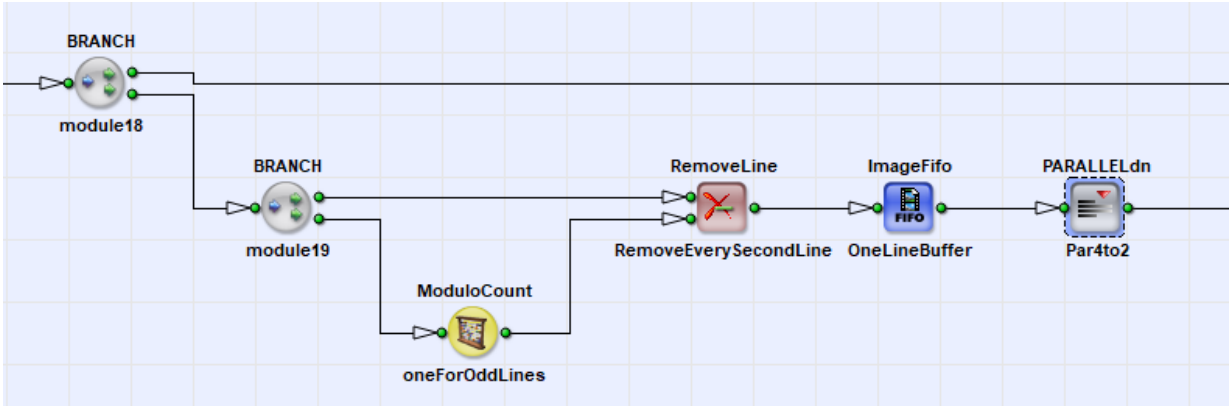



Figure 3.38. Bandwidth Limitation Compensated

## 3.6.8. P-Type Operators

P-type operators are similar to M-type operators, with one exception: They can not block the input actively.

Nevertheless, when a module further up the processing pipeline blocks its input, the input blocking is propagated backwards the line. If a P-type operator is positioned further down the line, the P-type operator`s input port gets blocked, too (passively). P-type operators propagate the blocking information backwards to the next module down the processing line.

**Example:** The M-type operator *PARALLELdn* requires more time to output the results compared to the time the data is fed into the operator. Therefore, it will block the input from time to time. In contrast, a P-type operator *PARALLELup* will always have a higher bandwidth at the output compared to the input bandwidth. This behavior is very important when connecting infinite sources with P-type and M-type operators. A detailed description of infinite sources is presented next in Section 3.6.9, 'Infinite Sources / Connecting Cameras'.

## 3.6.9. Infinite Sources / Connecting Cameras

Infinite sources are operators which cannot be stopped by the VisualApplets flow control. One example are the camera operators. On their input link, these operators receive data from the camera. On their output link, they forward this data to the successive module in the diagram. As there is no buffer within the camera operators, and as they are not able to tell the cameras to stop the transfer, these operators are **infinite sources**.

Infinite sources can only be connected to M-type operators which accept infinite sources. Whether an individual M-type operator can be connected to an infinite source or not is described in the respective documentation for this operator (see Part III, 'Operator Reference'). If an infinite source is connected to an M-type operator which does not accept infinite sources, the DRC level 2 will generate an error:

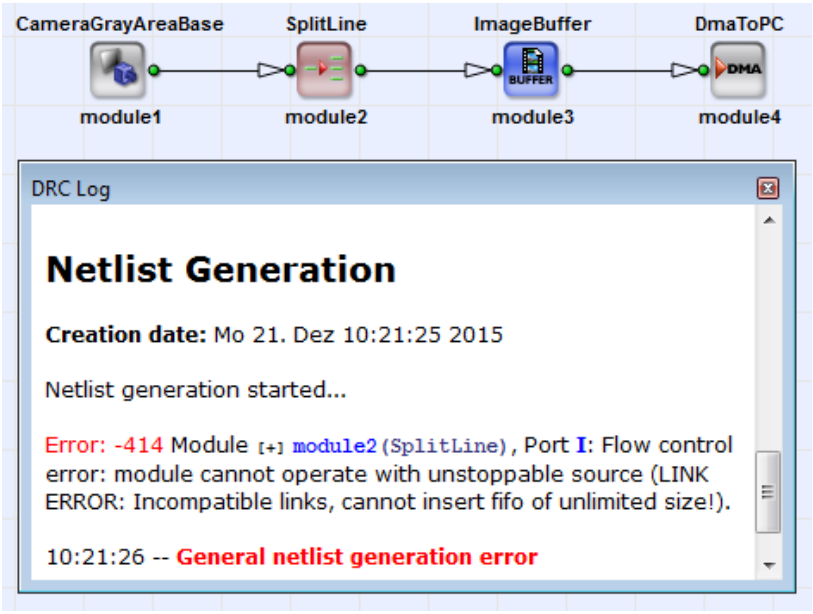



Figure 3.39. Infinite Source Connection Error

All P-type and O-type operators can be connected to infinite sources. The infinite source information will propagate through operators until an operator in the pipeline will convert the infinite source data stream into a controllable data stream. Usually, these conversion operators are buffers. A buffer operator is able to use the flow control at its output and buffers the input data stream without the need to block the input.

Thus, if we swap the operators of the previous example, the DRC level 2 will not generate an error:

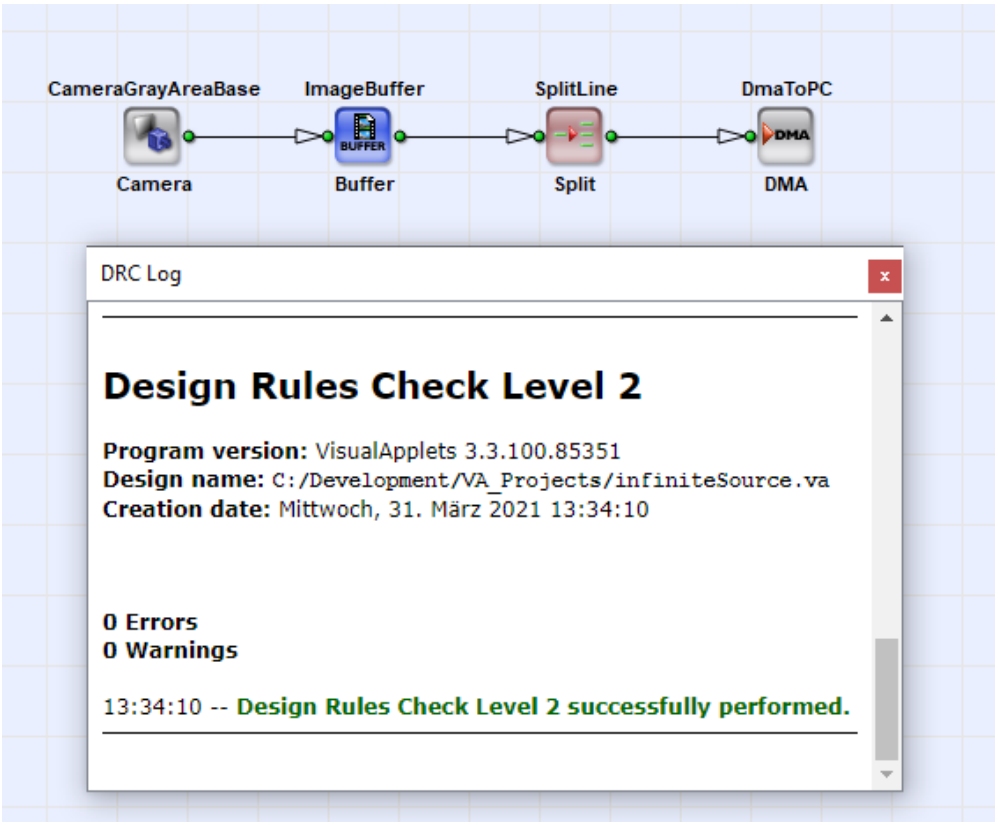



Figure 3.40. Infinite Source Connection OK

The behavior of some of the conversion operators can be changed to whether they accept infinite sources or not. These operators include a parameter called *InfiniteSource* which can be set to *Enabled* or *Disabled*.

A conversion operator which is not connected to an infinite source **must** be set to *Disabled*.

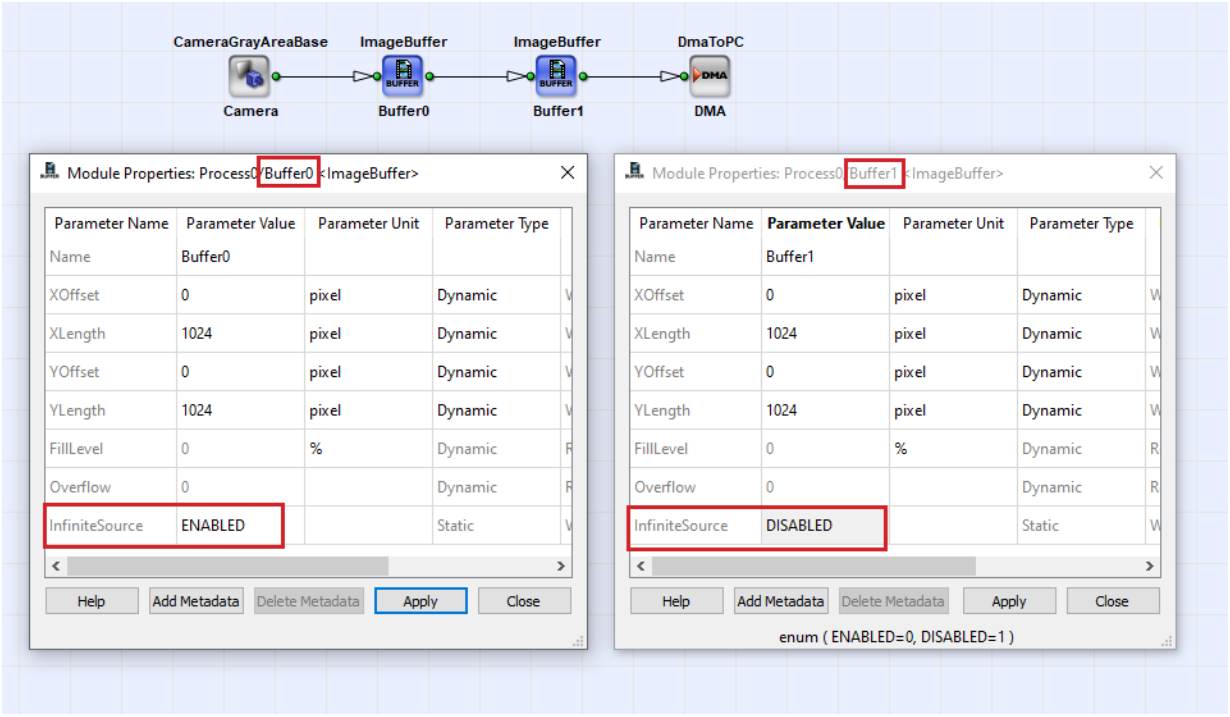



Figure 3.41. Infinite source conversion module (Buffer1) connected to a non-infinite source

**Warning**

If the parameter *InfiniteSource* of an respective module is set to *Enabled*, the module must be connected to an infinite source.

A conversion operator which is not connected to an infinite source must be set to *Disabled*.

**Tip**

Always set the parameter to *Disabled* if you are not sure, and change the failing modules after the DRC level 2 check.

For a detailed description of the parametrization of modules, see Section 3.7.1, 'Module Properties'.

## 3.6.10. Differing Rules for Signal Links

Signal links transport one-bit signal data which is valid at every clock cycle. The rules of links for modules with signal transporting input links differ in two ways from the rules of links described so far:

- All O-type and M-type modules with signal link inputs may be sourced by different M-type sources.
- Modules with signal link inputs may always be connected to infinite sources.

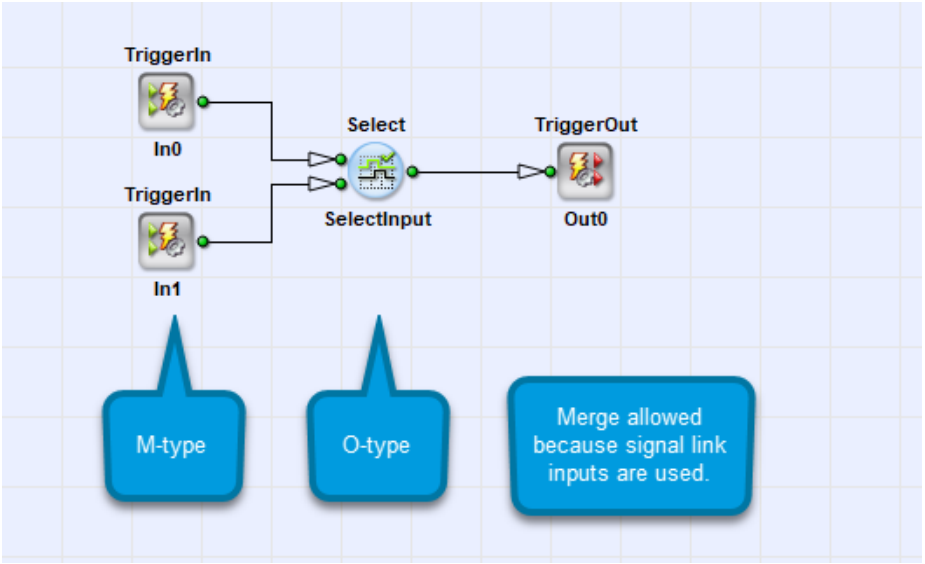



Figure 3.42. O-type module with signal link inputs, sourced by different M-type modules

Find more information on signal links in Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure'.

### 3.6.11. Summary

In this chapter you have been introduced to the rules of links. The rules of links are most important when implementing a VisualApplets application.

O-type networks always start and end with an M-type module. Each O-type module has to be sourced by the same M-type or P-type module through an arbitrary network of other O-type modules. Merges of different M-type sources can only be performed by M-type modules.

M-type networks may be arbitrarily linked in a serial sequence as well as they may be branched arbitrarily. Merging of different M-type sources in a module requires timing synchronization and image dimension synchronization. M-type modules with multiple inputs can have synchronous-input groups. Synchronous-input groups have to be sourced by the same M-type source through an arbitrary O-type network.

Timing synchronization will synchronize different sources which are timely delayed. Timing synchronization requires knowledge of the user implementation. A bad timing synchronization will result in deadlocks or buffer overflows.

An infinite source cannot be controlled by the integrated flow control. Only M-type operators which are allowed to be connected to infinite sources may be connected to infinite sources.

To modules with signal link inputs, the general synchronization rules apply with two modifications: The signal link inputs (of M-type **and** O-type modules) may be sourced by different M-type sources. Modules with signal link inputs may always be connected to infinite sources.



**Warning**

VisualApplets cannot detect logic synchronization errors as they depend on the user project implementation. Read the documentation of each M-type operator carefully to learn about its behavior in timing and delay.

## 3.7. Diagram Parametrization

In the previous sections, the data flow model, the layout and the rules of links of VisualApplets projects were explained. This chapter will outline the parametrization of diagram modules and links.

Application behavior and bandwidth do not only depend on the operators you use in your design, but also quite strongly on the parametrization of the modules and links. Module properties define the behavior of each module, while link properties describe the protocol between the modules. Module and link properties directly depend on each other and mutually influence their availability or ranges.

### 3.7.1. Module Properties

You can change module properties either in the *Module Properties* dialog or in the *Parameter Info* view. The *Parameter Info* view displays all parameters of all operators in the active design and thus provides a good overview of all configured parameters. The *Module Properties* dialog provides more details about the parameters. You can edit parameters in both views, however, adding metadata is only possible in the *Module Properties* dialog.

#### 3.7.1.1. The Parameter Info View

The *Parameter Info* is located in the *Information Panel* in the top right side of the VisualApplets main window.

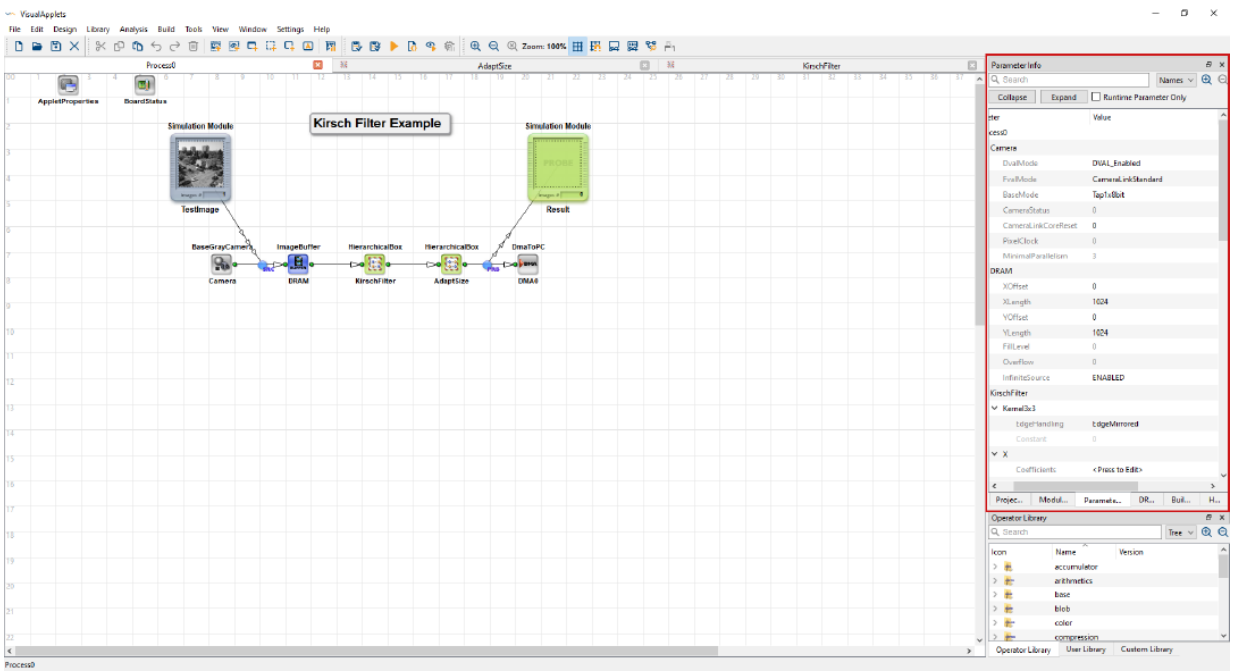
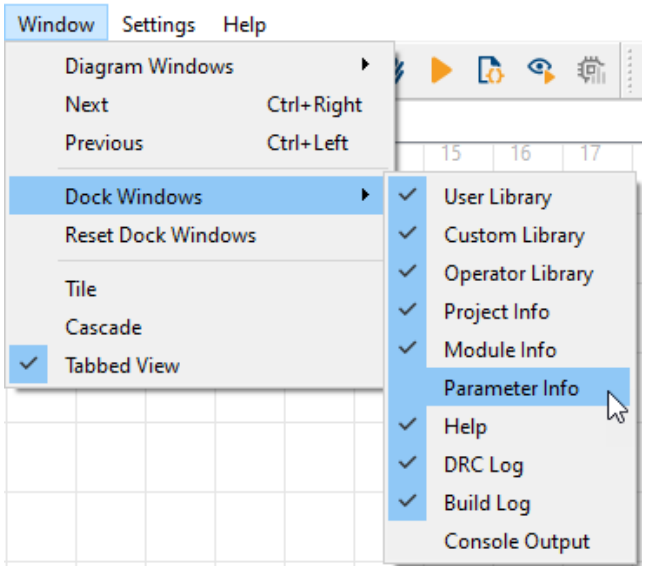



Figure 3.43. The Parameter Info View

If the *Parameter Info* doesn't open up at start, you can open it via the menu bar by selecting **Window** -> **Dock Windows** -> **Parameter Info**.




The *Parameter Info* view shows all parameters and their operators of the active design. You can also edit the parameters in the *Parameter Info* view.

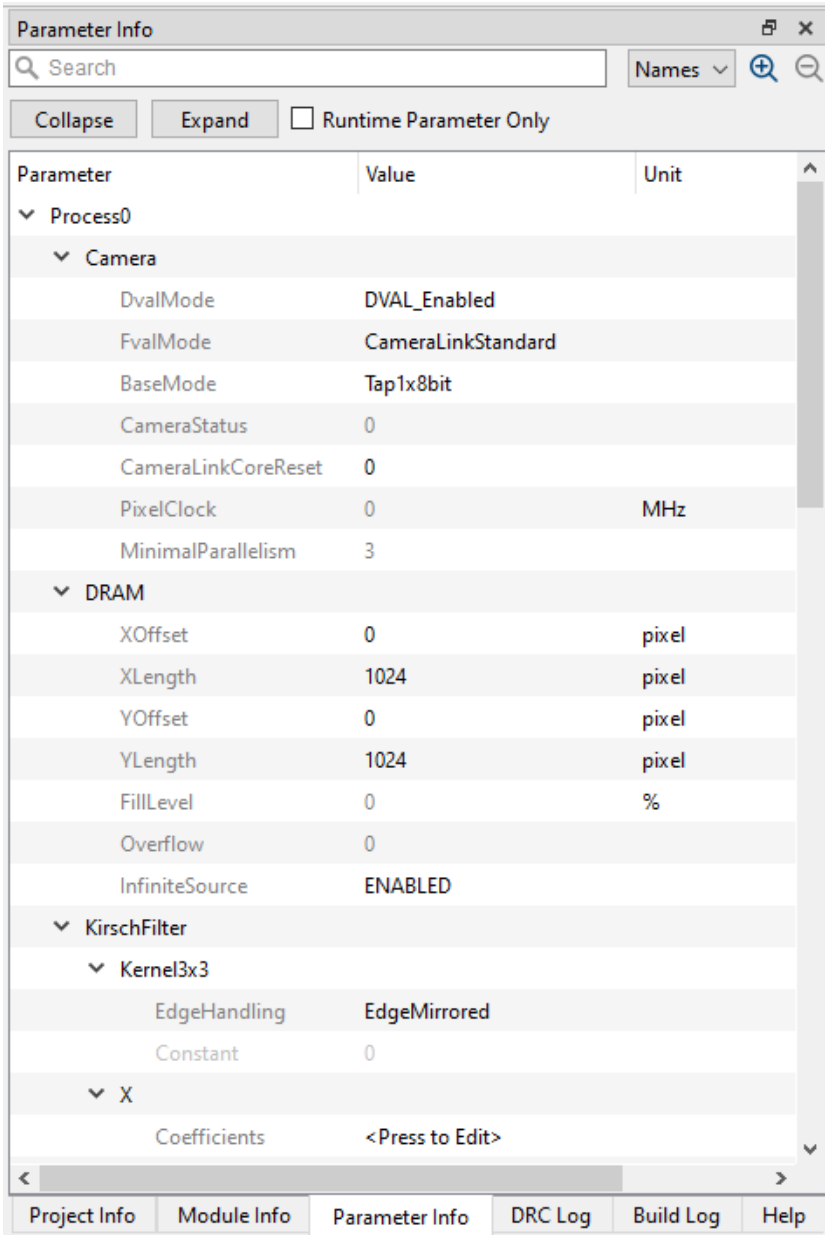
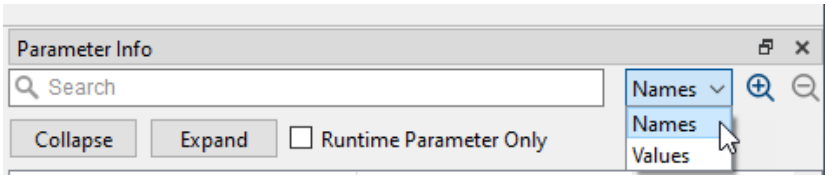



Figure 3.44. Parameter Info

The *Parameter Info* view provides a search function that allows you to search for parameters, parameter values, or operator names. You can use the *Runtime Parameter Only* filter to display only dynamical parameters, i.e. parameters that you can edit during runtime.




### 3.7.1.2. The Module Properties Dialog

1. To access the *Module Properties* dialog, double-click a module, or right-click on a module and select **Properties**.

The *Module Properties* dialog lists all required settings a module. Note that for each module a different set of parameters is available (depending on the operator that has been instantiated for creating the module).

The following figure shows the parameters of an *ImageBuffer* operator in the *Module Properties* dialog:

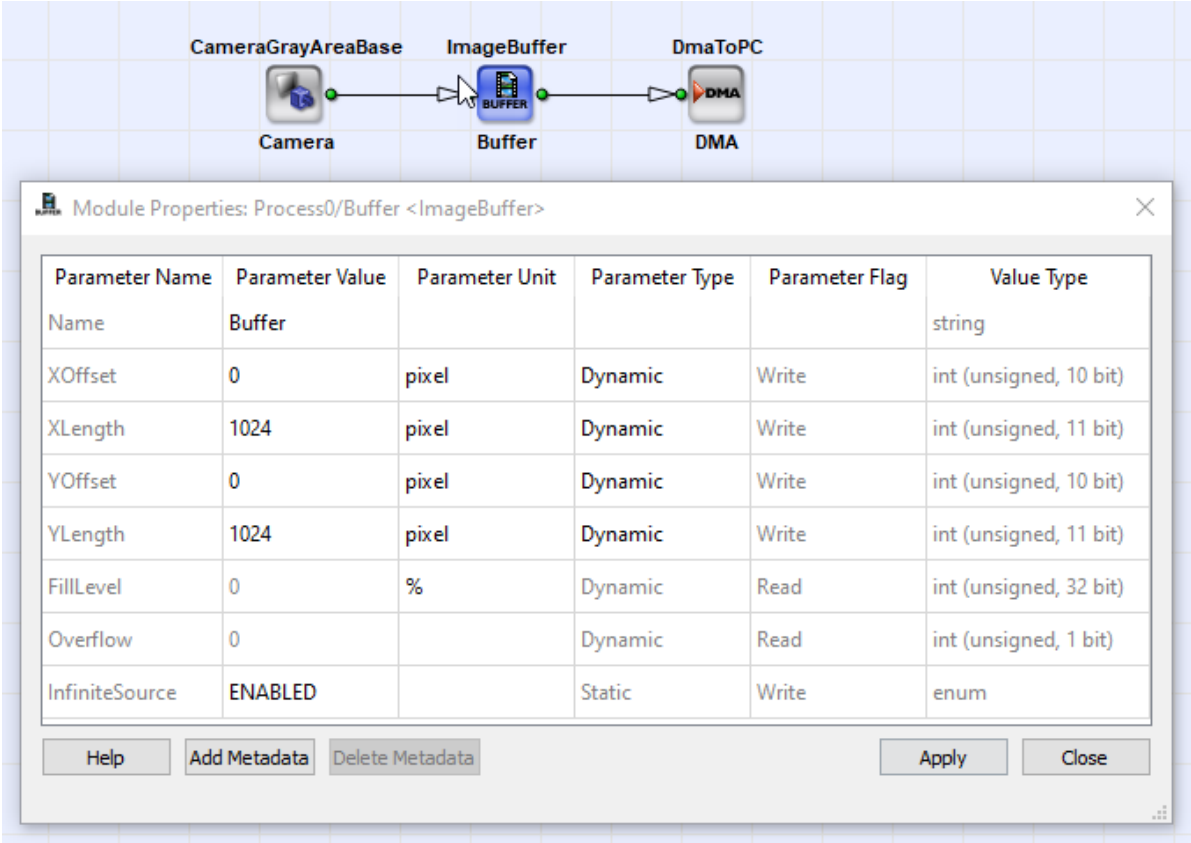



Figure 3.45. Module Properties Dialog

The *Module Properties* window lists all available parameters of a module. In this context, each parameter represents a module property. The parameters are displayed with entries in the following columns:

- *Parameter Name*:

  The name of the parameter.

  When adding a parameter to a module (see below), you can enter a name for the new parameter here.

- *Parameter Value*

  The actual value of an parameter.

  You can edit this field to change a value.

- *Parameter Unit*

  The unit of a parameter. Can be pixel, seconds, lines, etc.

- *Parameter Type*

  Here, you can set a parameter to static or dynamic. Static parameters cannot be changed after the build process of an applet, while dynamic parameters can be changed when the applet is already

in use on the frame grabber hardware. Mostly, static parameters will require less FPGA resources than dynamic parameters.

- *Parameter Flag*

  The flag of a parameter indicates whether a parameter is

  - read only (**Read**), or has
  - read & write access (**Write**).

- *Value Type*

  The type of the parameter value. There are four different types of parameter values:

  - String

    Any string, e.g., file names.

  - Values

    Values of different types (unsigned, signed, double) and ranges.

  - Enumerations

    A list of enumeration values such as *Disabled* or *Enabled*.

  - Field Parameters

    A field parameter consists of an array, a list, or a matrix of values. It is mostly used for kernel coefficients, lookup tables and other lists. Value ranges and size of list can be dynamic. Note that the displayed bit width is not equal to the actual parameter range. Moreover, a parameter is always unsigned, even if it is declared as a signed value in the column.



**Differences Between Modules**

How many parameters and which parameters are available for a certain module is different from module to module (depending on the underlying operator). Nevertheless, each module will possess at least one parameter called *Name*.

### 3.7.1.3. Parameter Editing



**No Undo option**

It is not possible to cancel a change. All changes are immediately applied.

To edit a parameter:

1. Click on a parameter value and type in the new value.

   Some value types have special editing options:

   **String:** A string parameter you can edit by simply typing in the new string. Some string parameters are file name parameters. In this case, a file selection dialog will open.

   **Values:** For editing a value, use the spin box next to the parameter value to increase or decrease the value. The step size of the parameter defines the minimal increase/decrease.

   **Fields:** With field parameters, a new window will open for editing which shows all field elements of the parameter. The following figure shows the fields of a *FIRoperatorNxM* operator:

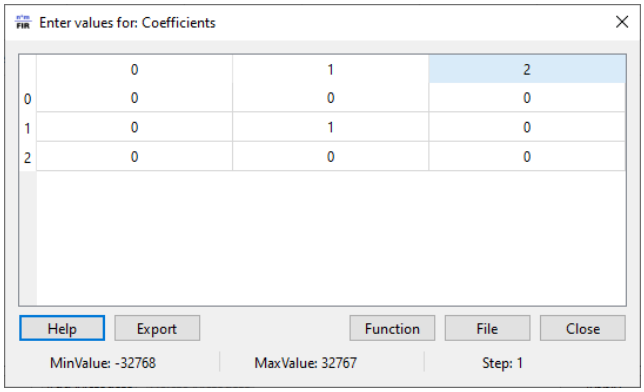



Figure 3.46. Field Parameter Edit Window

There are three different ways you can edit the field parameter values of a module:

a. You can simply edit each field element individually.

b. Alternatively, you can use a list from file to fill the field parameter values.

- You can import the file by clicking on the **File** button.
- You can also export the current field parameter setting to file by clicking the **Export** button.



**File Format of Import File**

- The import file has to contain a list of ASCII values.
- Each value has to be stated in a separate line.
- The lines have to be separated by carriage return (CR) and line feed (LF).
- The file has to be in text only format ([NameOfFile].txt).

If the file contains more values than there are in the field, the first values of the file are used (as many as are required to fill the field). The following values in the file have no effect.

If the file contains less values than there are in the field, all values of the file are used exactly one time. The last positions of the field (for which there are no values in the file) will remain unchanged.

c. A third possibility to edit the elements of a filed is the *Parameter function* dialog.

- Click on **Function** to open the function dialog (see next figure).

  Using this dialog, a linear function can be used to edit the elements. A linear function consists of the elements $a$ = slope, $b$ = offset, and $x$ = index of field element. In the dialog, $a$ and $b$ can be edited.

- Enter values for a and b.

  The first element of a filed has index 0. For matrix elements, the elements are consecutively numbered row by row.

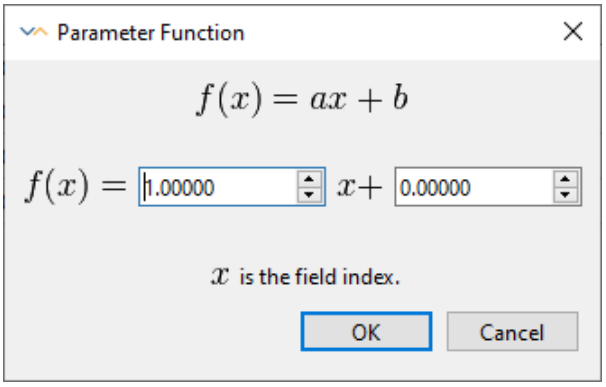



Figure 3.47. Function Dialog to Edit Field Parameters

**Disabled Parameters**

Some parameters are disabled and cannot be edited. Whether a parameter is disabled or enabled depends on the settings of other parameters, the input format, or the output format. Check the respective operator reference for more information. The following screenshot shows two disabled parameters. In this case, the y parameters are disabled because the input link has image protocol VALT_LINE1D (this means, the image is one-dimensional (just a line of pixels) and thus, there is no y-coordinate).

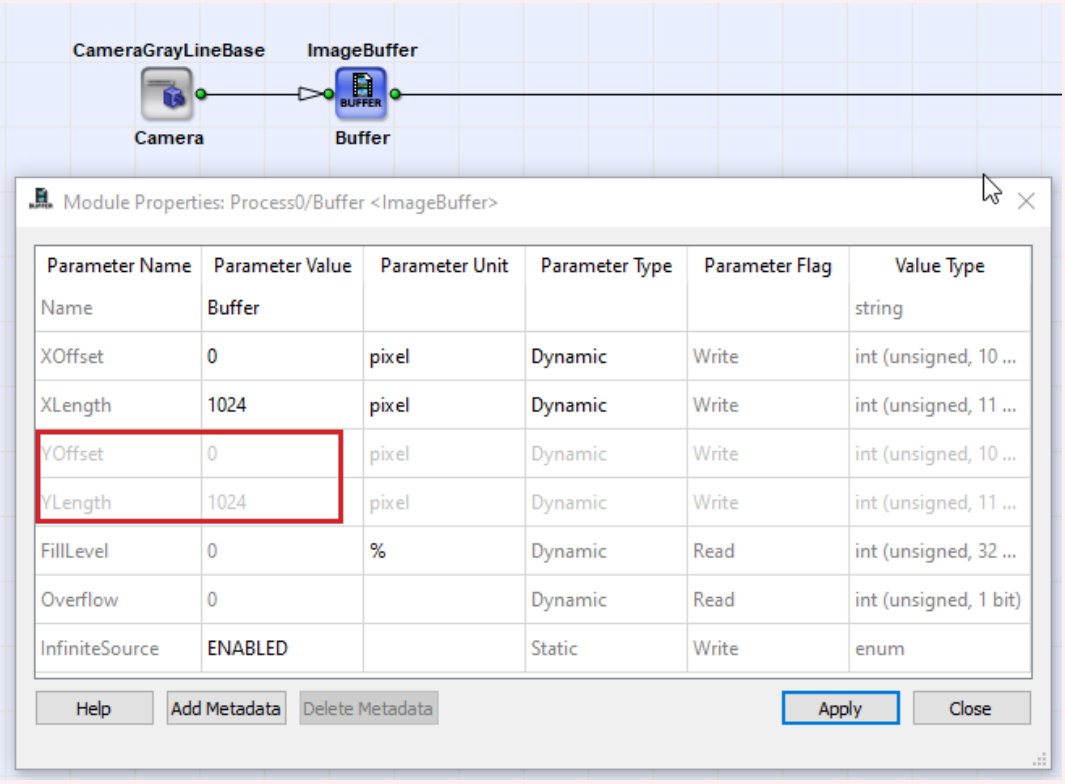



Figure 3.48. Disabled Parameters

After a parameter has been edited, it has to be applied.

2. Click on **Apply** in the *Module properties* window to apply the settings.

A new parameter value is discarded if it is not within the allowed parameter value range.



**Click Apply before closing the window**

It is recommended to click on Apply before closing the parameter. This way, you will see warnings if the value you entered is not accepted.

3. Close the *Module properties* window by clicking on **Close**.

### 3.7.1.4. Autocompletion and Syntax Highlighting for Translator and Reference Operators

Most operators of the Parameters library provide an autocompletion and syntax highlighting functionality for specific parameters. Autocompletion is available for the following parameters in the following operators:

| Operators | Parameters | Activation |
|---|---|---|
| *EnumParamTranslator* | *WriteAction*, *ReadAction* | Typing "${" |
| | *DisplayHierarchy* | **TAB** key |
| *FloatParamTranslator* | *WriteAction*, *ReadAction* | Typing "${" |
| | *DisplayHierarchy* | **TAB** key |
| *IntParamTranslator* | *WriteAction*, *ReadAction* | Typing "${" |
| | *DisplayHierarchy* | **TAB** key |
| *LinkParamTranslator* | *WriteAction* | Typing "${" |
| *EnumParamReference* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *FloatFieldParamReference* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *FloatParamReference* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *IntFieldParamReference* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *IntParamReference* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *ResourceReference* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *StringParamReference* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *IntParamSelector* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *FloatParamSelector* | *Reference* | **TAB** key |
| | *DisplayHierarchy* | **TAB** key |
| *EnumVariable* | *DisplayHierarchy* | **TAB** key |
| *FloatVariable* | *DisplayHierarchy* | **TAB** key |
| *IntVariable* | *DisplayHierarchy* | **TAB** key |

Table 3.2. Availability of Autocompletion and Syntax Highlighting

To use the autocompletion for a reference operator, click into the *Parameter Value* field of the *Reference* or *DisplaHierarchy* parameter, and press the **TAB** key. VisualApplets then offers you a selection of valid values in a drop-down list box.

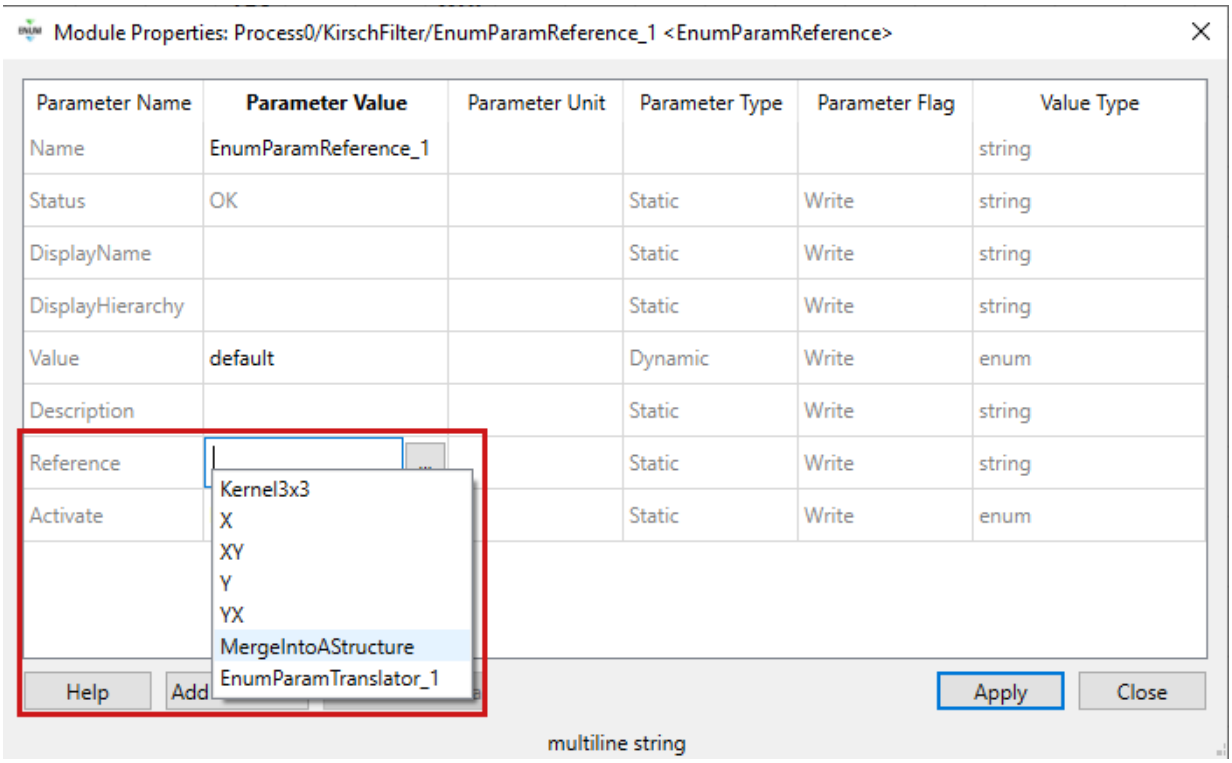



Figure 3.49. Autocompletion for Reference Parameters

If you click the three dots ... next to the *Parameter Value* field, the *Enter data for <operator>/<parameter>* dialog opens up. In this dialog, you see the data with syntax highlighting and can further use the autocompletion feature:

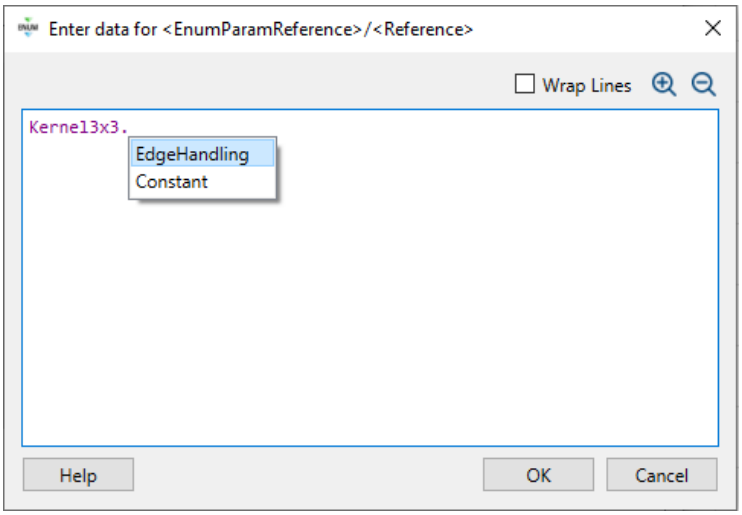



Figure 3.50. Autocompletion in "Enter data for <operator>/<parameter> Dialog

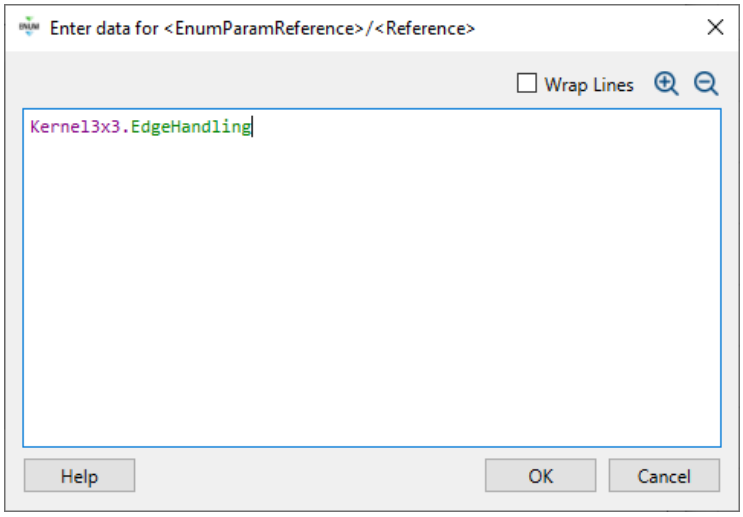



Figure 3.51. Syntax Highlighting in "Enter data for <operator>/<parameter> Dialog

Syntax highlighting also works in the *Module Properties*:

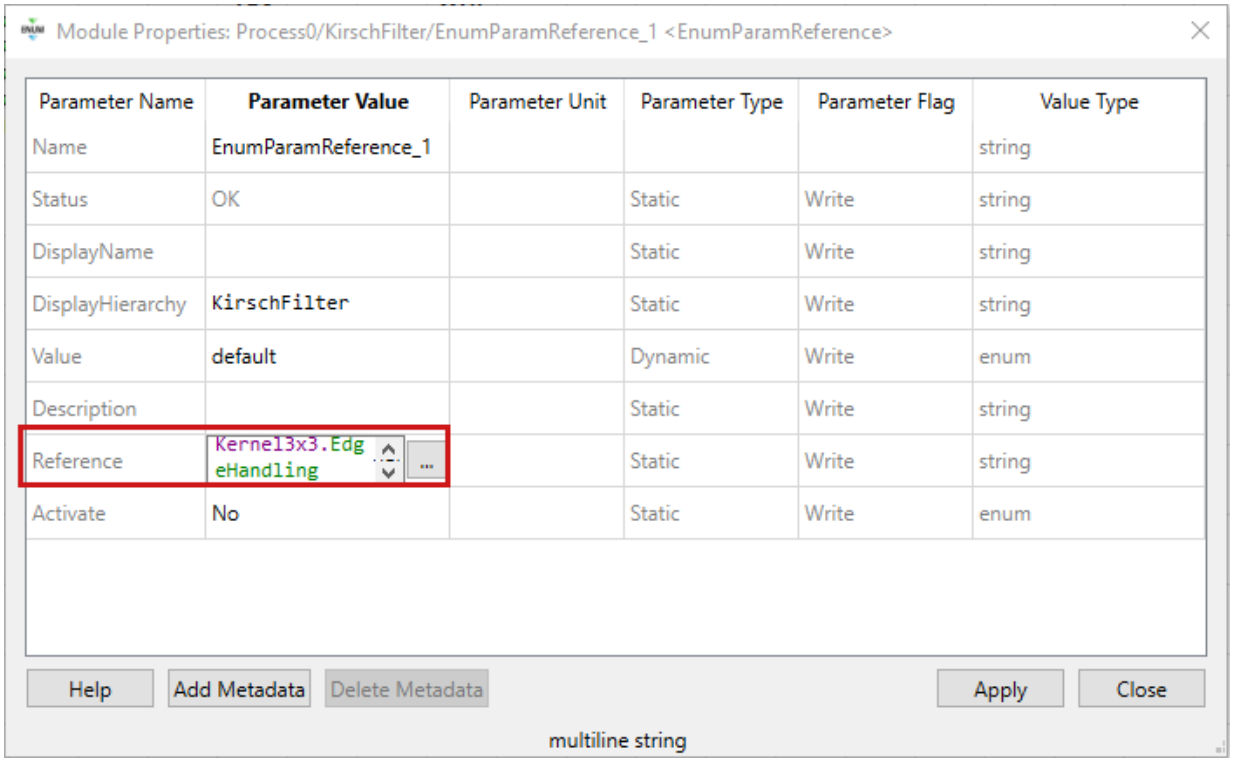



Figure 3.52. Syntax Highlighting in Module Properties Dialog

To use the autocompletion functionality in the *WriteAction* and *ReadAction* parameters of translate operators, type "${" into the *Parameter Value* field:

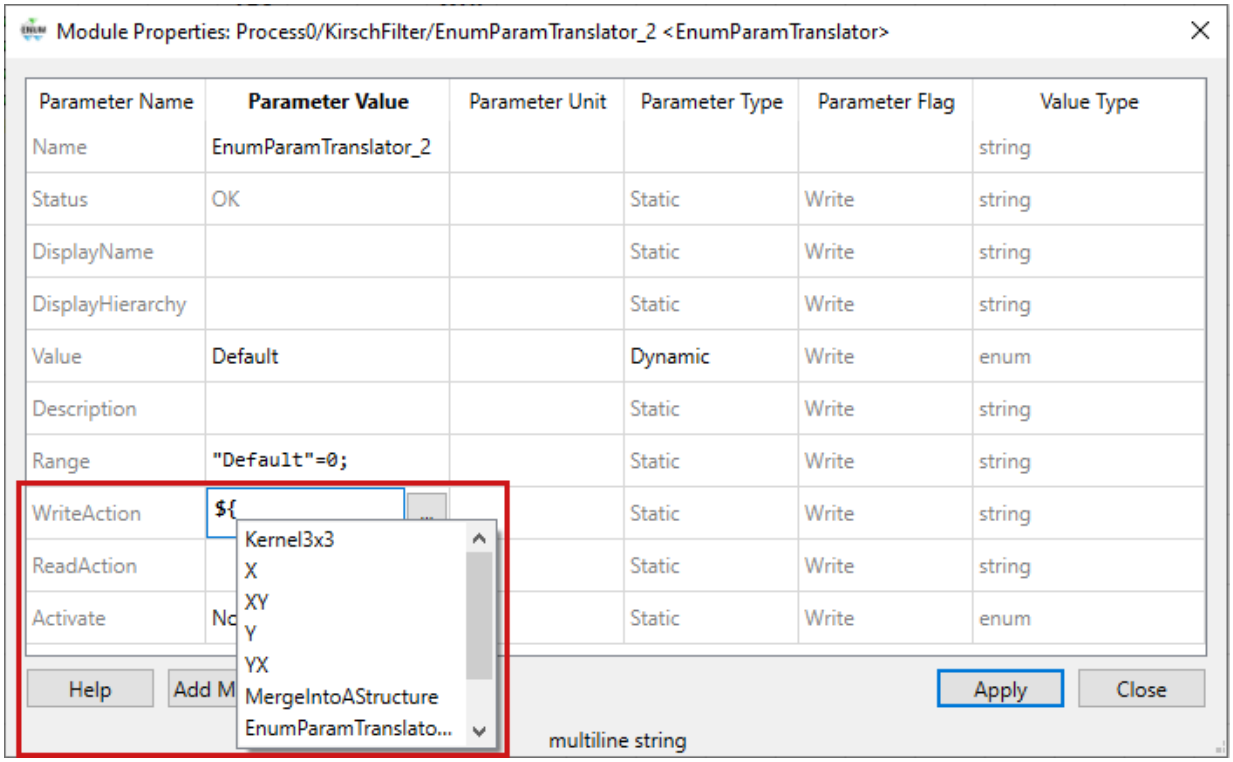



Figure 3.53. Autocompletion for Translator Operators

Syntax highlighting also works for the *WriteAction* and *ReadAction* parameters of translator operators.

To use autocompletion for the *DisplayHierarchy* parameter of translate operators, click into the *Parameter Value* field of the *DisplaHierarchy* parameter, and press the **TAB** key.

### 3.7.1.5. Illegal Parameter Value States

The allowed value range of a parameter might change when link properties or by the setting of other properties. If this happens, the old parameter value might be in an illegal state. In this case, the parameter is marked red, to indicate, that the current setting will not pass the design rule check. You have to correct the parameter value until it is in the valid range.

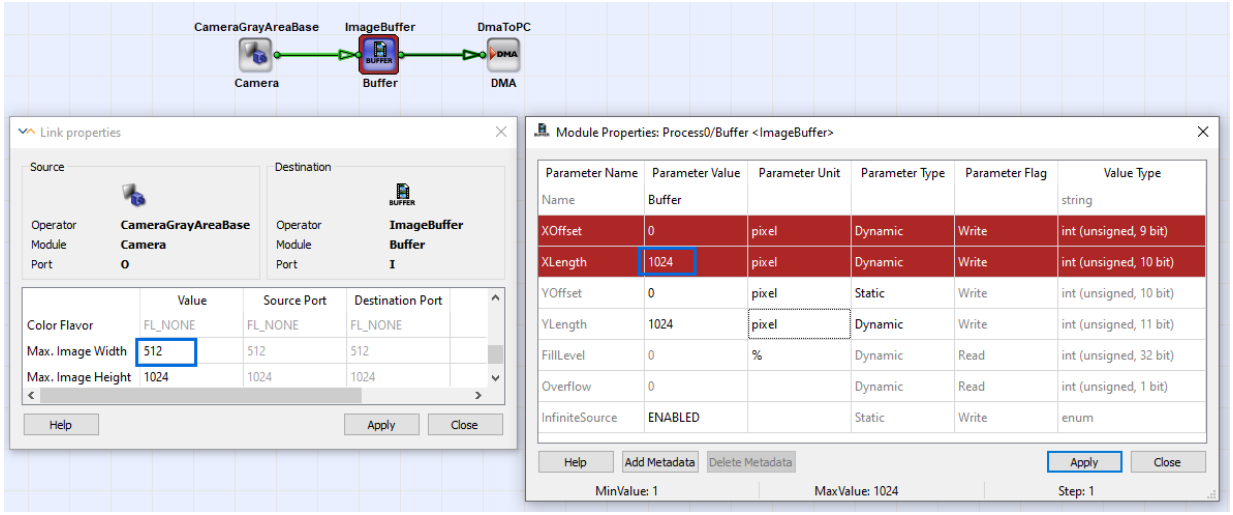



Figure 3.54. Parameters in Illegal States

The previous figure showed an example of parameters in an illegal state. In this case, the parameter value is not within the allowed range, as the *XLength + XOffset* is greater than the parametrized maximum image width of the input link.

### 3.7.1.6. Metadata

For each module, metadata parameters can be added. Metadata parameters are string parameters. To add a metadata parameter click on the **Add Metadata** button in the module properties window. A new parameter is now added to the parameter list. By clicking the name and the value, both can be edited. Metadata parameters are read / write parameters. To delete a metadata parameter, simply select the parameter and click on **Delete Metadata**.

Metadata parameters are often used for versioning and commenting.

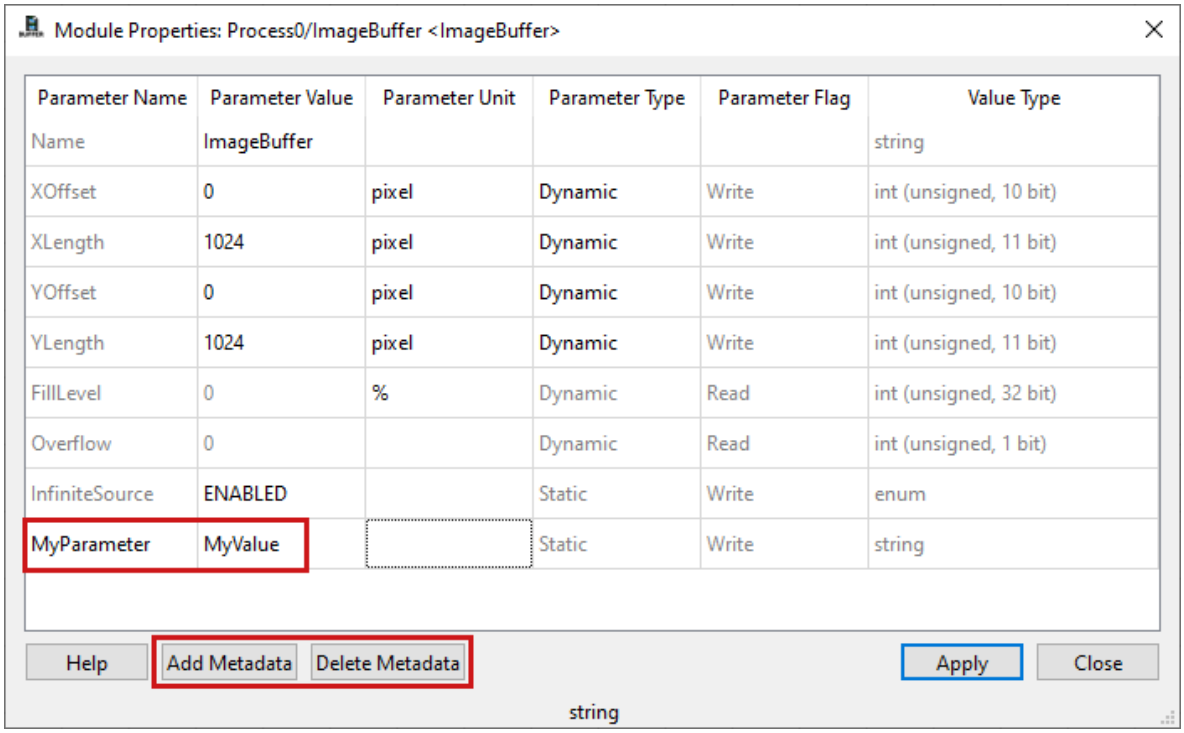



Figure 3.55. Metadata Parameter

## 3.7.2. Link Properties

Each link between modules is defined by its link properties. You access link properties via double click on a link or via right click and selection **Properties**.

Each link has the following link properties:

- *Bit Width*

  Defines the bit width of the link for one pixel. Thus if color links are used, the bit width represents all color planes. For *image protocol* VALT_SIGNAL the bit width is always one bit.

- *Arithmetic*

  The arithmetic of a link. Can be signed or unsigned. If set to signed, a two's complement signed arithmetic is used. The sign bit is included in the bits specified with property *bit width*. For *image protocol* VALT_SIGNAL the arithmetic is always set to unsigned.

- *Parallelism*

Specifies the number of pixel which are transferred in parallel between two modules within one design clock cycle. For *image protocol* VALT_SIGNAL the parallelism is always set to one. More information about the parallelism can be found in Section 3.5.1, 'Bandwidth of an Applet'.

- *Kernel Columns*

  Defines the number of kernel columns. For *image protocol* VALT_SIGNAL the number of kernel columns is always set to one.

- *Kernel Rows*

  Defines the number of kernel rows. For *image protocol* VALT_SIGNAL the number of kernel rows is always set to one.

- *Image Protocol*

  The image protocol defines which protocol to transfer data is used by the link. The following protocols are available:

  - VALT_IMAGE2D: Two-dimensional images, e.g., area scan cameras.
  - VALT_LINE1D: One-dimensional images, i.e., images with an unlimited image height, e.g., line scan cameras.
  - VALT_PIXEL0D: A simple stream of pixels with no image dimension information.
  - VALT_SIGNAL: One bit signals which are valid at every clock cycle. Used for signal processing such as trigger systems.

  More information on image protocols is presented in Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure'.

- *Color Format*

  The color format can either be VAF_GRAY, VAF_COLOR or VAF_NONE. For *image protocol* VALT_SIGNAL the color format is always set to FL_GRAY.

- *Color Flavor*

  Defines the flavor of the color format. For *image protocol* VALT_SIGNAL the color flavor is always set to FL_GRAY.

- *Max. Image Width*

  Defines the maximum image width of a link. Images transported on the link must not exceed this value but may be less than the value. The *Max. Image Width* has no influence on links with *Image Protocol* VALT_PIXEL0D and VALT_SIGNAL. For these image protocols the *Max. Image Width* is ignored. More information can be found in Section 3.5, 'Data Flow ' and in Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure'.

- *Max. Image Height*

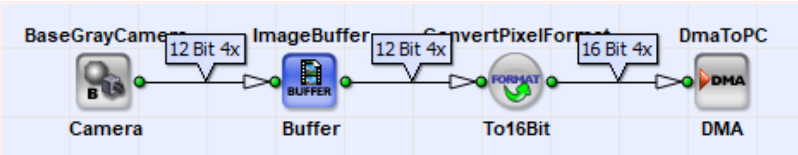
  Defines the maximum image height of a link. Images transported on the link must not exceed this value but may be less than the value. The *Max. Image Height* has no influence on links with *Image Protocol* VALT_LINE1D, VALT_PIXEL0D and VALT_SIGNAL. For these image protocols the *Max. Image Height* is ignored. More information can be found in Section 3.5, 'Data Flow ' and in Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure'.



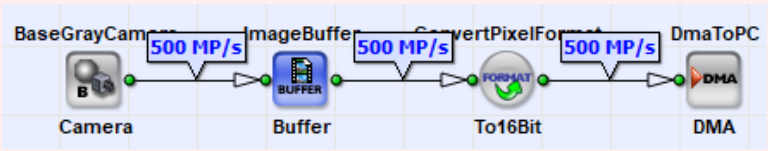
**Visualization on GUI**

For visualization of the according link properties, VisualApplets provides two GUI buttons in the toolbar of the program window:

**Display Link Info**  displays the bit width and the parallelism for every link in the diagram:




**Display Link Throughput** displays the maximum pixel throughput in megapixels per second for every link in the diagram:




### 3.7.2.1. Properties Ranges and Disabled Properties

Each module defines the ranges of the link properties at its output itself. Some properties cannot be modified, some properties can only be modified in a specified range while others can arbitrarily be modified. A detailed explanation of the allowed output link formats is given in the respective operator reference in Part III, 'Operator Reference'.

### 3.7.2.2. Parameter Editing

To edit a parameter, simply click on a parameter in the "Value" column of the link properties window.

The link properties dialog consists of three columns:

- Value

  This is your editing column. Edit your link property values in this column by clicking on the values. If the edited value is not within the allowed parameter range or step size, the edited value is discarded or cannot be inserted. Confirm the new value by leaving the focus of the field or by pressing the Enter key.

- Source Port

  After a new value has been entered in the Value column and has been confirmed, the source operator checks if the new value is accepted. If the new value is not accepted, the parameter and the link will get a red color to show a link property conflict. Thus, if a link property in column source port is dyed red, the source operator does not allow the settings. **The values in this column cannot be modified!**

- Destination Port

  A click on **Apply** or **Close** or **Window Close (X)** or pressing **Esc** will apply the new settings for the link. If the new settings are not accepted by the destination operator input port, the link as well the parameter in column Destination Port will be dyed red to indicate the conflict. **The values in this column cannot be modified!**

A change of a link property can cause the change of another link property or a module parameter value. Canceling changes is not possible. Each change will immediately be applied.

Let's have a look at some examples. The following figure shows a conflict at the source port. The source operator, in this case a *CastParallel* operator, does not accept a parallelism of three at its output because of the bit width. After the output parallelism has been changed to four, two or one, the conflict is solved.

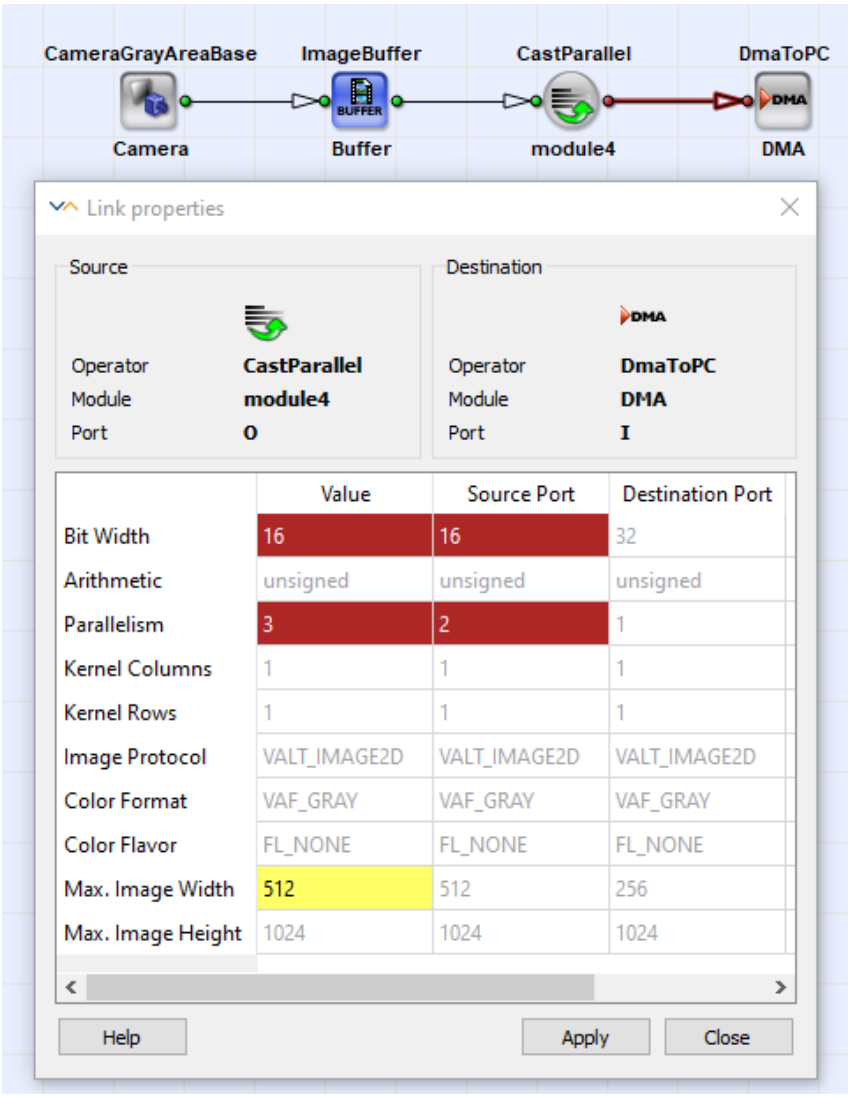



Figure 3.56. Invalid Source Port Link Properties

Screenshot Figure 3.57, 'Invalid Destination Port Link Properties' shows a conflict at the destination port. The destination operator, in this case a *DmaToPC* operator, does not accept RGB images with this parallelism and bit width. After the parallelism has been reduced to four or five, the conflict is solved.




Figure 3.57. Invalid Destination Port Link Properties

### 3.7.3. Propagation and Dependencies of Operator Parameters and Link Properties

The link properties are defined by the operator. Some link properties can directly be changed at the operator output links, while others are fixed or are defined by the operator parameters. Moreover, most operators define their link properties in addition from the input link properties. This can result in a link property modification chain through multiple modules. Thus, if a link property has been changed, the parameters and link properties of successive module might change, too.

This propagation chain can result in three results:

- The modification has been accepted by all successive modules and links. VisualApplets will display all links which have been changed in green, to inform the user about modified link properties.
- The link modification is not accepted at the link of some module. In this case, check the link for the error. Read the documentation of the source or destination operator to learn about the allowed link formats.

- The modification causes a conflict with a module parameter. In this case, the module is dyed red as well as the respective module parameter. Check the operator reference for link and parameter dependencies to resolve the conflict.

## 3.8. Allocation of Device Resources

Applets and included modules require resources. The required resources differ for each project. Mostly, the FPGA internal resources are used. An explanation of the internal resources is given in Section 3.11, 'FPGA Resource Estimation'.

Besides the FPGA internal resources, applets require external frame grabber hardware resources as well as logic resources. Some of these resources are listed and allocated in the *Device Resources* dialog window, which you can open via **Design** -> **Device Resources**.

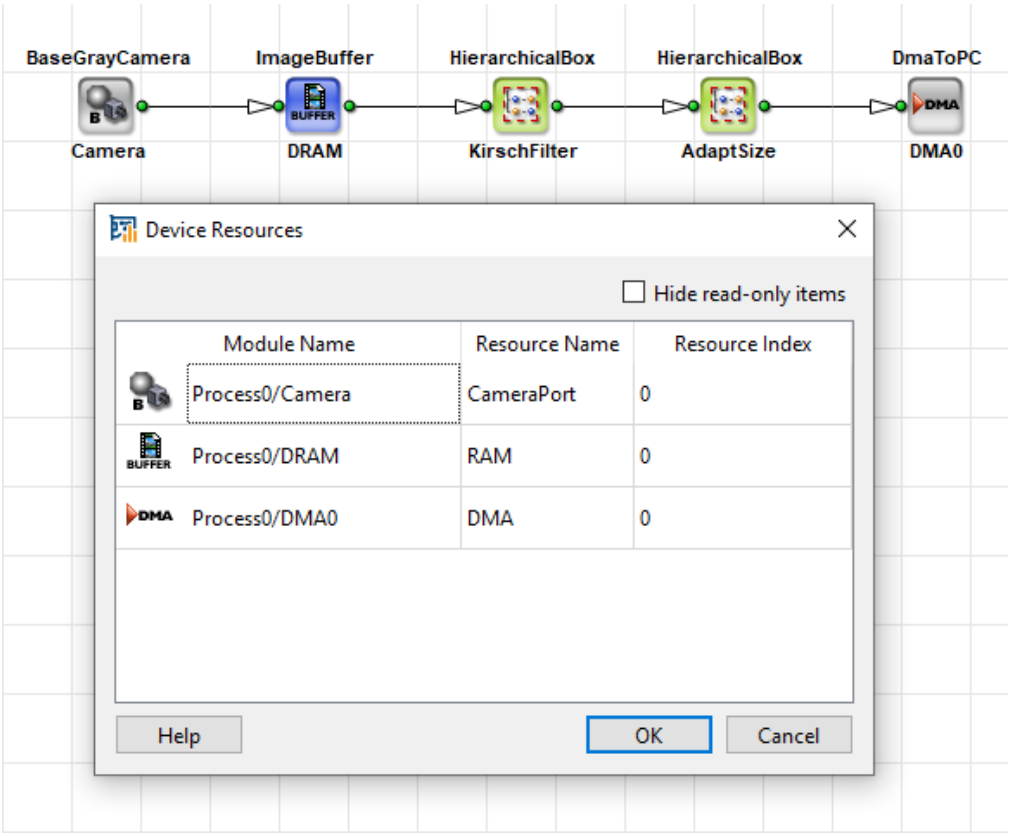



Figure 3.58. Device Resource Allocation Window

The following table lists all types of resources.

| Type | Description | Listed in *Device Resources* Window |
|---|---|---|
| CAM | The camera port index | yes |
| RAM | Index of *frame grabber RAM* | yes |
| DMA | The DMA channel index. Both, DmaToPC and DmaFromPC operators use this resource. | yes |
| CameraControl | For Camera Link frame grabbers: The camera port index of the four camera control outputs. | yes |
| TriggerOut | Frame grabber digital outputs | no |
| TxLink / RxLink | Links between a pixelPlant hardware and the frame grabber. | no |
| Signal Channel (operators TxSignalLink/ RxSignalLink) | Used for arbitrary signal links which are not shown in the diagram. | no |

| Type | Description | Listed in *Device Resources* Window |
|---|---|---|
| Image Channel (operators TxImageLink/ RxImageLink) | Used for implementing loops into a design. | no |
| Event | Logic Event Name (Bit). | no |
| EventSource | Event source e.g. an operator. | no |

Table 3.3. List of Device Resources

### Display of Device Resources

Not all of these device resources are listed in the *Device Resources* window. Some of them are automatically assigned, while others are set in the modules which use the device resource.

Some operators for microEnable 5 marathon designs use resources that are displayed, but cannot be set in the *Device Resources* dialog since they are controlled by the operator. These resources are grayed-out in the dialog:

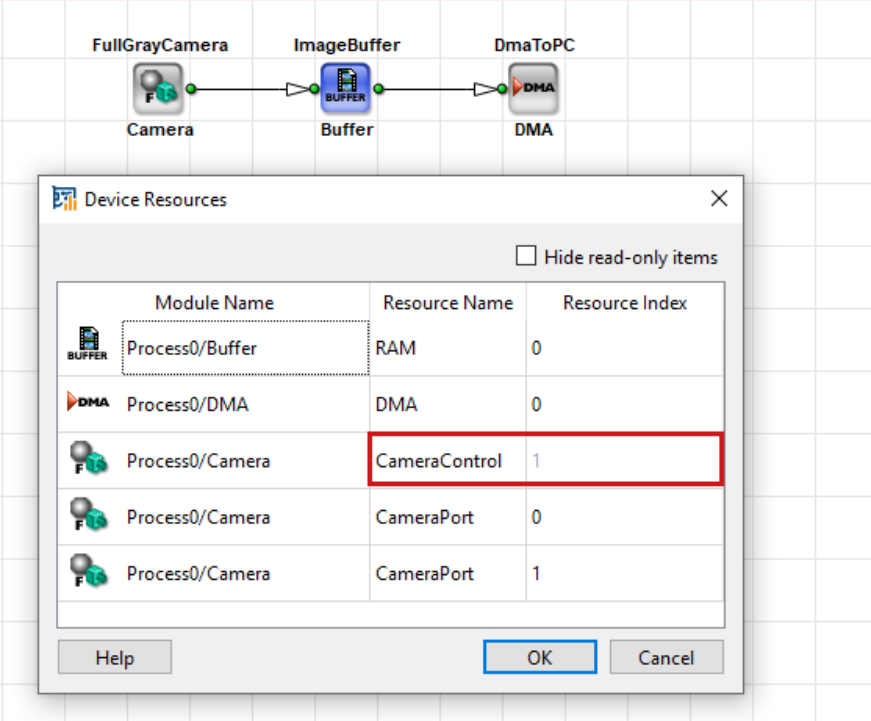



Figure 3.59. Grayed-out resource CameraControl

All resources indices must only be used once. Moreover, some resources have to be in ascending order, starting from zero. If these conditions are not given, the design rules check will fail and the allocation has to be changed. The *Device Resources* window marks conflicts in red. Change the allocation until all conflicts are solved. The following figure shows an example.

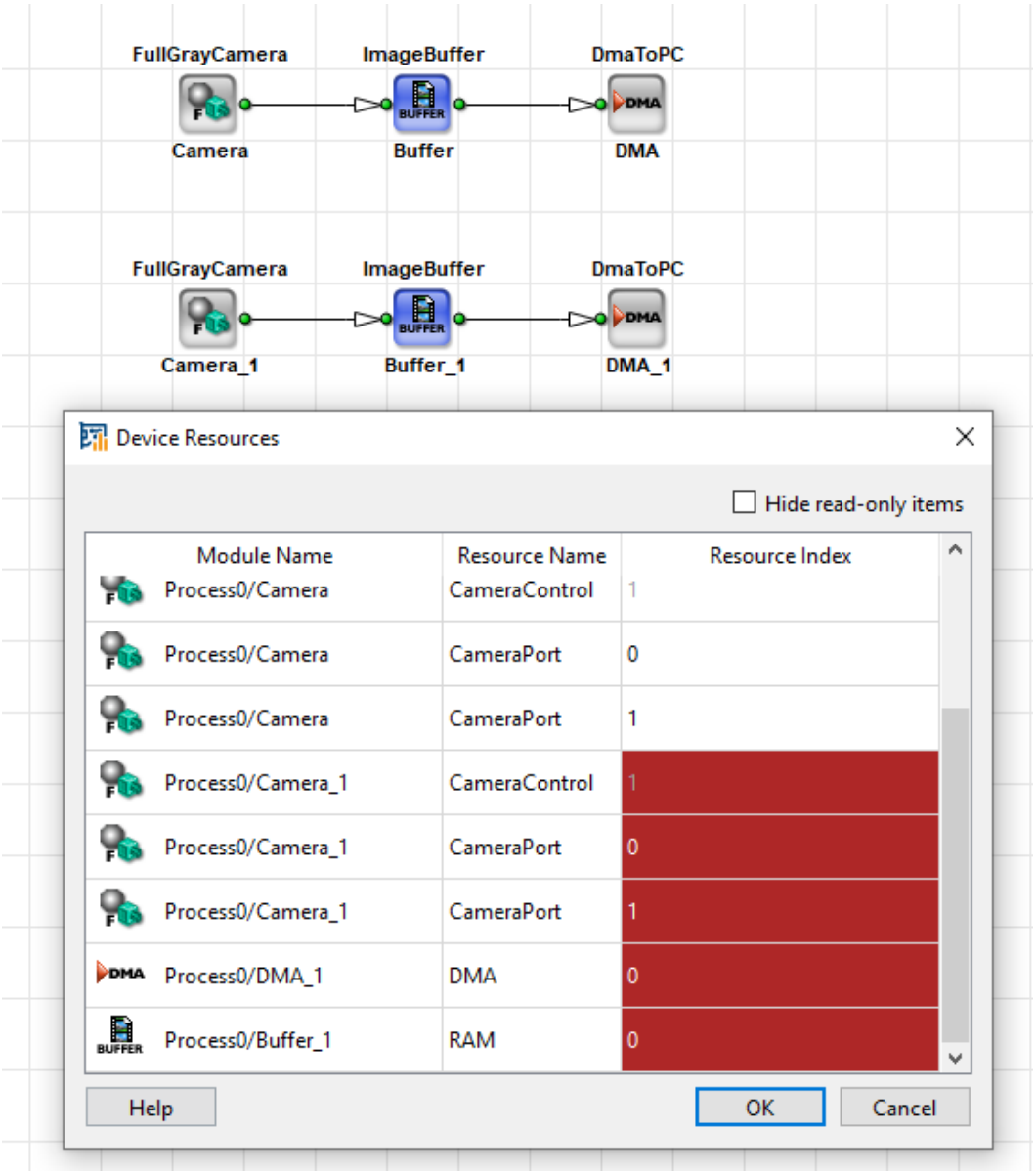



Figure 3.60. Device Resource Conflict

Instead of manually solving the conflicts, VisualApplets can automatically solve conflicts. Click on **OK** in the *Device Resources* window. If a remaining conflict exists, VisualApplets will automatically resolve the conflict. You can either click on **Apply** to accept all changes without verification and close the window. Or you can click on **Show Details...** to see what has been changed (next figure). If you do not want to accept the changes, click on **Retry** to undo changes and go back to the *Device Resources* window.

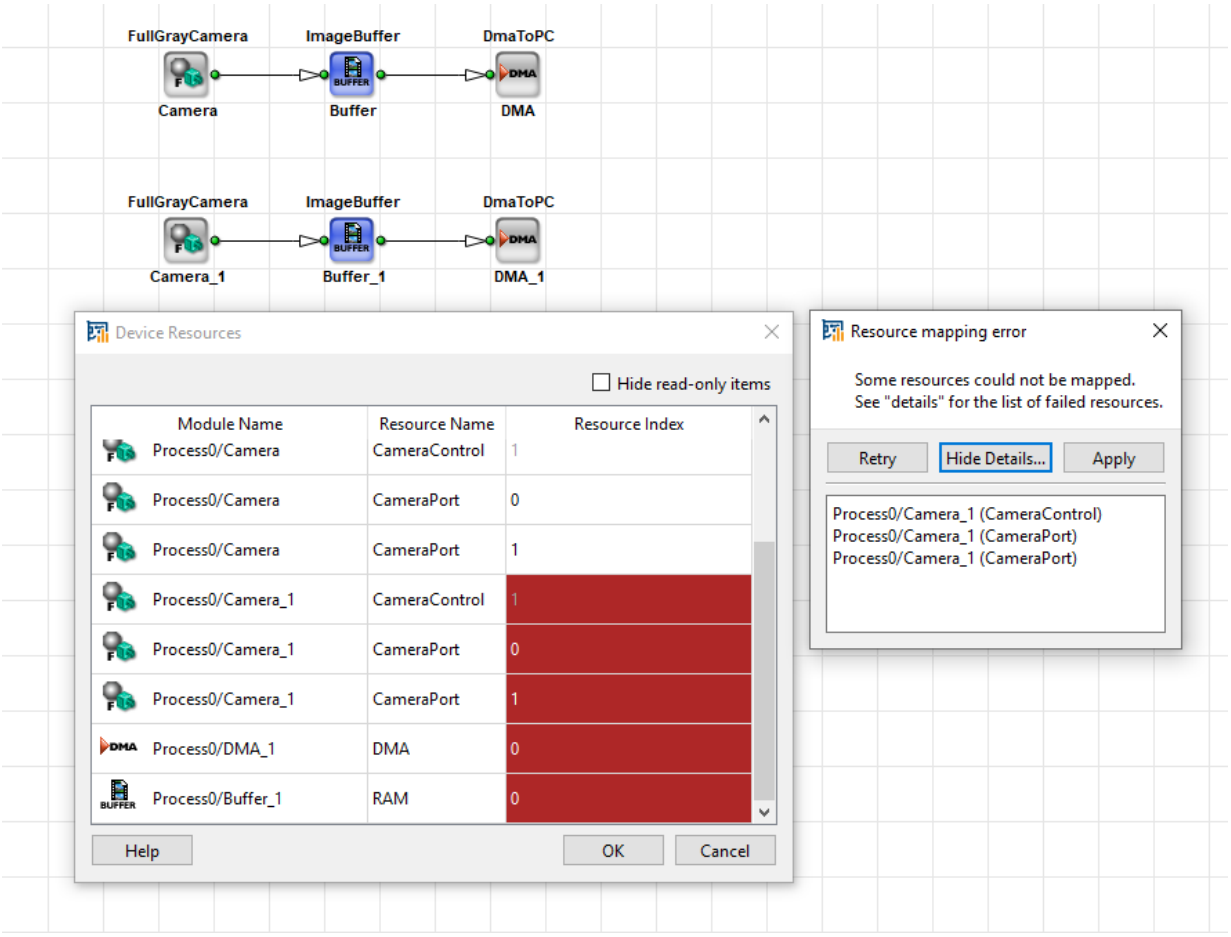



Figure 3.61. Auto Correction of Device Resource Conflicts

Please note that an automatic change might not always result in the desired solution. For example, an automatic change might change the camera allocation.

**List of Device Resources for all Supported Hardware Devices**

In Appendix A, '*Device Resources*' a detailed list of the available device resources for all supported hardware devices is listed.

Device resources are used for interfacing the FPGA environment (Camera, DRAM etc.)

# 3.9. Design Rules Check

The design rules check (DRC) is an elementary function of VisualApplets. A design rules check checks if a VisualApplets project does not contain any design errors. Projects which do not pass the DRC can not be build or some functions are disabled.

The DRC log window which is part of the dock, lists the progress and results of the DRCs. It shows important information on design errors and resource usage estimations. Some windows include a DRC log as a sub-window. The DRC log can be saved in a HTML file using a right click in the text and the selection of **Save to file** on the pop-up menu.

In VisualApplets includes two levels of design rules checks. DRC level 1 and DRC level 2 which are explained in detail in the following sections.

## 3.9.1. DRC Level 1

The DRC level 1 checks the integrity of the project. Click on **Analysis** -> **Design Rules Check Level 1** ( **Ctrl+F7** ) or use the icon Design Rules Check Level 1 from the Build icon bar to start the DRC level 1. Besides the direct access, the DRC level 1 is automatically started by other VisualApplets functions such as simulation, resource estimation, or build process. The DRC level 1 checks amongst others whether

- no open ports exist in a design.
- no link is in conflict state.
- no operator parameter is in conflict state.
- no conflicts for device resources exist.

The following figure shows four errors reported by the DRC level 1. In this example, three device resource conflicts of the resources CAM, RAM, and DMA are shown as well as a resource index conflict of Cam0 and Cam1. You can simply click on the module name to highlight the module in the diagram which causes the error. This is very useful in complex projects with many diagram windows.

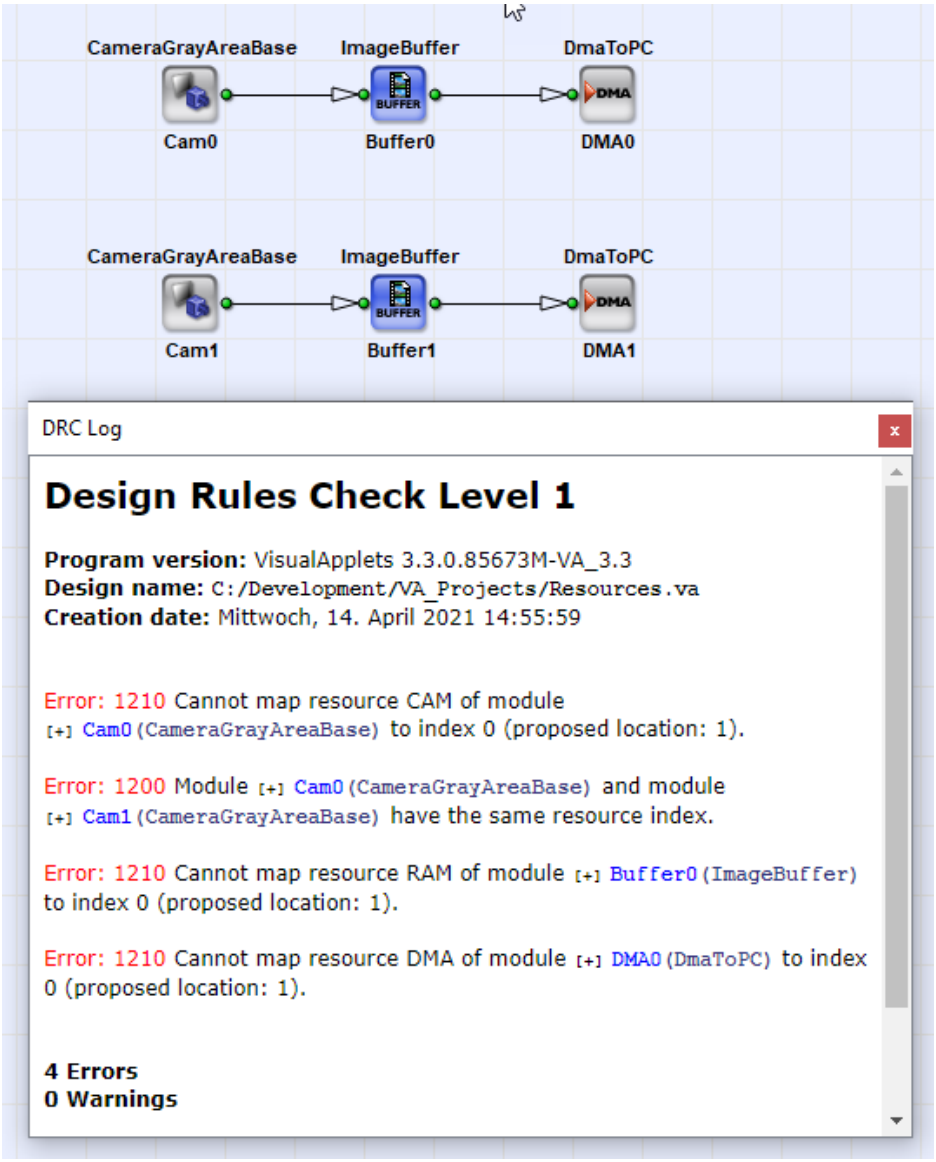



Figure 3.62. DRC Level 1 Error

### 3.9.2. DRC Level 2

The DRC level 2 can only be started after a successful DRC level 1. However, starting the DRC level 2 by a click on **Analysis** -> **Design Rules Check Level 2** ( **Ctrl+F8** ) or a click on the icon Design Rules Check Level 2 from the Build icon bar will automatically start DRC level 1, too. Besides the direct access, the DRC level 2 is automatically started by other VisualApplets functions such as the resource estimation and build process. The DRC level 2 performs a low level design rules check and estimates the required FPGA resources.

## 3.10. Simulation

VisualApplets provides powerful, functional simulation features for simulating designs. You can use simulation for a first test of the implemented image processing algorithm, and for its verification. You can start a simulation directly after editing the design without the need to build a *.hap file.

The simulation in VisualApplets is functional, i.e., it emulates the behavior of the hardware implementation. This makes it very fast compared to low level simulations. A side effect of the functional simulation is that timing is not considered. The simulation behavior of operators is 100% equal to the image processing of the final applet on real hardware.

To simulate the behavior of a design in VisualApplets, two kinds of simulation elements are provided:

- simulation sources, and
- simulation probes.

Simulation sources are used for data input. You can load one image or a whole image sequence into a simulation source (from image files). You can place one simulation source or one simulation probe at any link in your design. Thus, data transport along those links is overwritten by the connected source module or inspected by the respective probe module.



**Note**

Some operators can function as an image source, too; e.g., operator *CreateBlankImage*.

You can save the resulting images of the simulation probes to image files.

Due to visualization optimizations, the VisualApplets simulation is based on 2D images.

Nevertheless, you can also simulate if the link the simulation source is connected to is 0D (stream of pixels), 1D (stream of lines), or 2D (stream of frames), as long as the simulation source contains at least one 2D image (see Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure').

### 3.10.1. Limitations

Simulation has the following limitations:

- The simulation of the SIGNAL image protocols is not possible. This is so, because the simulation of the SIGNAL image protocols emulates the functionality, but not the timing of a design. However for simulation of the SIGNAL image protocols, it would be necessary to simulate the timing, too.
- The simulation of the 0D image protocol is limited to exactly one simulation step. This is so, because for simulation all pixel data of one source is aggregated to a single 2D frame. However, subsequent steps can be simulated for the remaining design, but operators generating 0D content will only emit data once during the first step.
- The operator *PseudoRandomNumberGen* does not return the same result in a simulation as on the hardware. This is so, because the operator is non-deterministic. This exceptions is also described at Part III, 'Operator Reference'.
- The operator *Blob Analysis 1D* does not return the same result in a simulation as on the hardware. This is so, because the operator has the port FlushI, but as the flush is asynchronous to the image data, it cannot be simulated to reflect the same behavior as in hardware. This exception is also described at Part III, 'Operator Reference'.
- BMPs and TIFFs with indexed color palette are not supported.

### 3.10.2. Supported Image Formats

The following image formats are supported for simulation in VisualApplets for source images as well as for saving the output images:

| | |
|---|---|
| **Bitmap** | BMP 1 bit black/white (row length has to be a multiple of 8) |
| | BMP 8 bit gray |
| | BMP 24 bit (R8 G8 B8) sRGB color space |
| | Compression like RLE is not supported. |
| **TIFF (Tagged Image File Format)/TIF** | TIFF 1 bit black/white (row length has to be a multiple of 8) |
| | TIFF 8 bit gray |
| | TIFF 24 bit color sRGB color space |
| | TIFF 48 bit color sRRB color space |
| | Compression: None, LZW, or PackBits |

**JPEG/JPG**

**PNG**

**GIF**

**PSD**

Additionally, the following formats are supported as source images:

- SVG
- XCF
- RAW
- RSD (SiSoRawSimulationData)



**Limitations**

BMPs and TIFFs with indexed color palette are not supported.

## 3.10.3. Simulation Workflow

Simulating image data processing in your design comprises the following steps:

1. Inserting the simulation sources and probes you need into your design.
2. Loading the image file(s) you want to use for simulation to your simulation source(s).
3. Optimizing your image input via parameters, such as pixel merge and pixel alignment.
4. Setting the number of processing cycles for the simulation in the main simulation window and starting the simulation.
5. Evaluating the simulation results.
6. Saving the simulation results.

In the following sections, these steps are be described in detail.

## 3.10.4. Inserting Sources and Probes into your Design

To prepare a simulation, you must first insert your simulation source(s) and probe(s) into your design:

1. From the main menu, select **Analysis** -> **New Simulation Source / New Simulation Probe** or use the corresponding icons in the toolbar to insert your source(s) and probe(s) into the current design window.

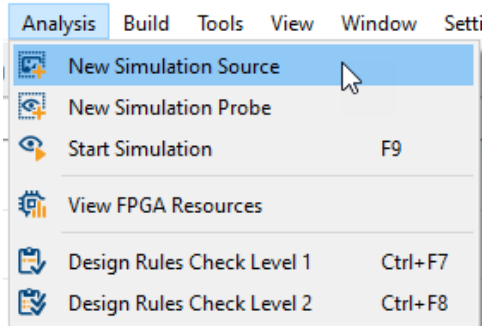



Figure 3.63. Creating New Simulation Sources and Probes

Simulation sources and simulation probes are both displayed as small image frames with a magnetic anchor:

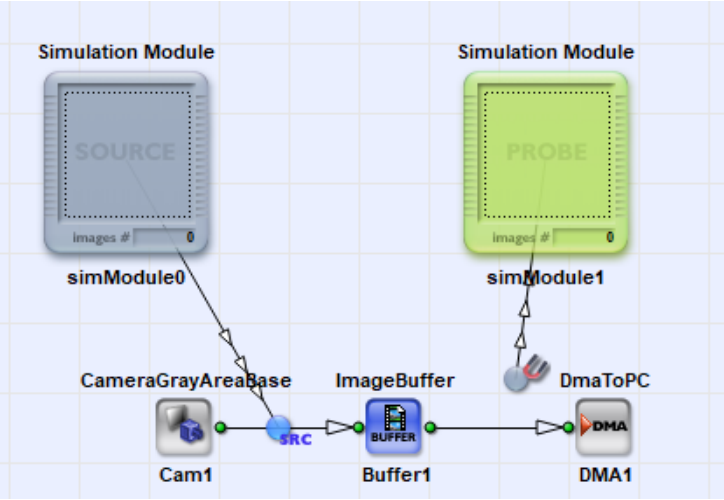



Figure 3.64. Simulation Sources Are Gray Image Frames, Simulation Probes Are Green Image Frames

2. Use the anchor to connect the source or probe to a link in the design. You can easily distinguish simulation sources from simulation probes as sources are of gray, probes of green color.
3. Use drag and drop to position your source(s) and probe(s).
4. Use drag and drop to connect your source(s) and probe(s) to links in your design. If the anchor icon changes to blue, the connection is valid.

**Connecting Simulation Sources and Probes to a Link Transporting Kernels or Signals**

You can connect a simulation source to a link that transports kernels, but the simulation source needs to provide exactly one input image per kernel. You can connect a simulation probe to a link that transports signals, but this simulation probe then doesn't provide any data.

## 3.10.5. Loading the Image File(s) to your Simulation Source(s)

### 3.10.5.1. The Simulation Source Viewer

In the **Simulation Source Viewer**, you can load test images or test image sequences and make them available for simulation.

At first, let's have a quick look at the **Simulation Source Viewer**.

1. To open the **Simulation Source Viewer**, double-click a source in your design.

The **Simulation Source Viewer** opens up:

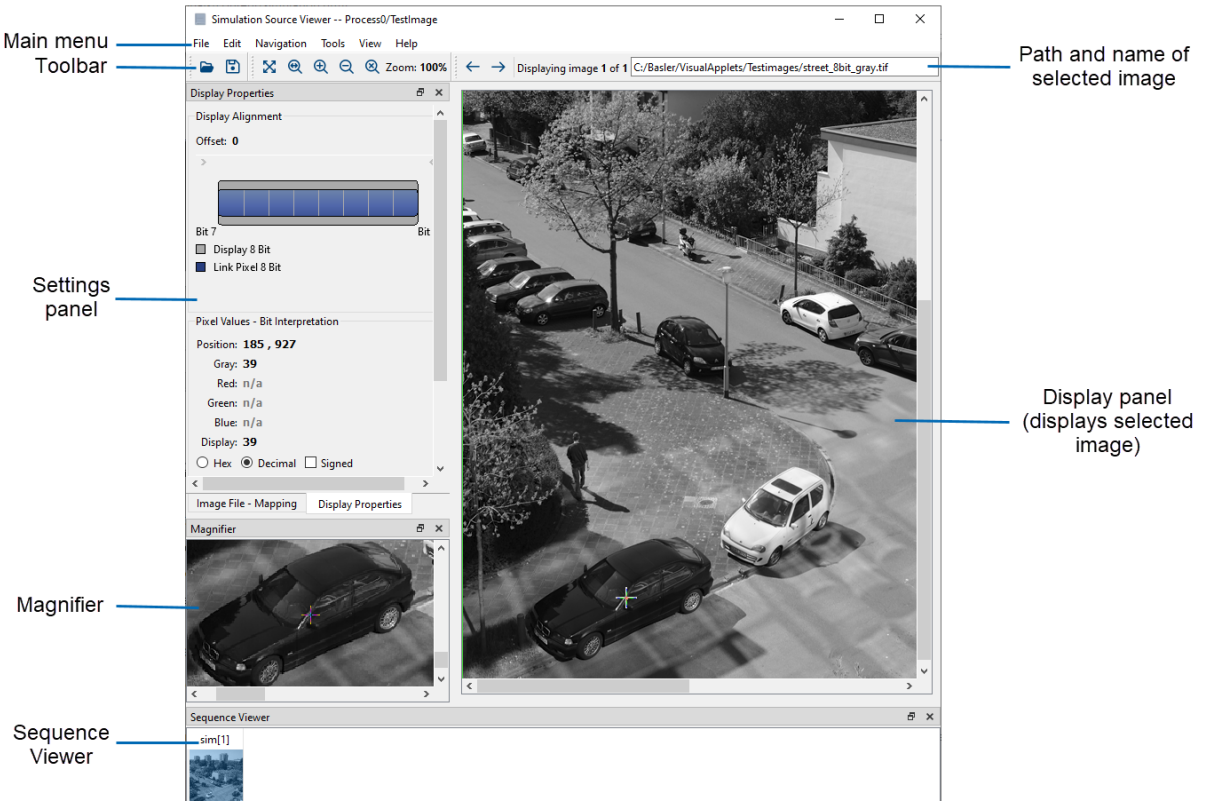
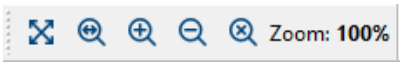



Figure 3.65. **Simulation Source Viewer**

Directly under the menu, you find the toolbars. The icons of the toolbars offer the following options (left to right):

-  File toolbar for opening and saving image files.
-  View toolbar offering different options for image display. These are the same as the first options of the View menu:

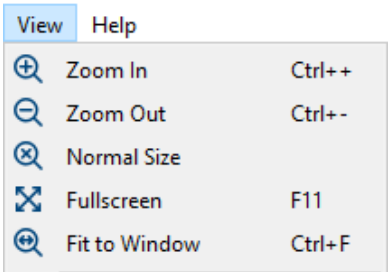
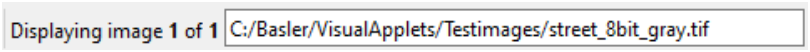



Figure 3.66. Viewing Options

-  The arrows for navigating through image sequences.

- 

Display of details on the image currently visible in the display panel of the main viewer window (and magnifier): Displays the position of the image in the image sequence loaded to the simulator, and the path to the image file

On the left hand side, you have the settings panel with two tabs:

**Image File – Mapping** Here, you can adjust the settings for simulation, e.g. enter a value for pixel merge, or define the offset for pixel alignment.

**Display Properties** Here, you can define the settings for displaying the test images on your own screen. Due to the fact, that monitors always use 8 bit color display, at times you have to decide which part of a pixel you want to see on screen for evaluating your test images. None of the settings you define under **Display Properties** have influence on the simulation or its results.

In addition, you find here the color values of a selected pixel listed, together with the display value. The display value is the mapped value for display on the user monitor.

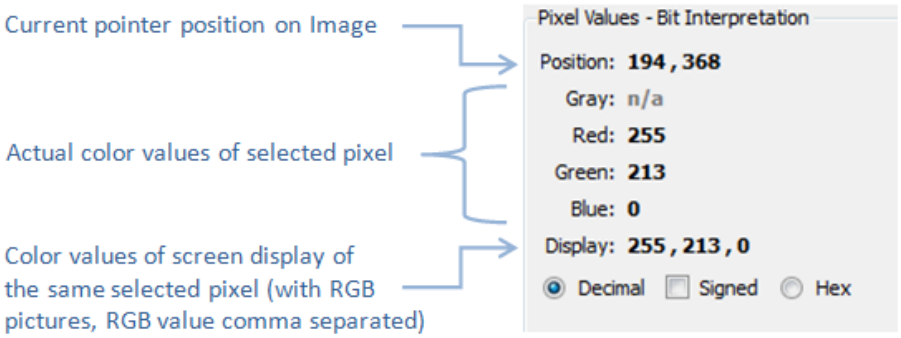



Figure 3.67. Pixel Values

On the right hand side, you have the display panel displaying the image currently selected.

You can zoom in and out on the image displayed in the display channel by either using the corresponding icons in the toolbar, or by using **CTRL +MouseWheelUp** to zoom in and **CTRL+MouseWheelDown** to zoom out.

If you choose a very high zooming factor, a pixel grid is displayed for better orientation and hex/dec values per pixel are displayed.

On the bottom of the simulation viewer, you have another panel: This is the **Sequence viewer**. If you use more than one image in a source, here you can see all images of the image sequence, make changes to the image order of the sequence, or select an image for display in the display panel and magnifier.

The **Magnifier** is actually a second window of the **Source Viewer** which you can easily loosen from the main **Source Viewer** window via drag & drop. The **Magnifier** always shows the same image as is displayed in the display panel of the main **Source Viewer**. Its pointer is always in the center of the **Magnifier** window and positioned exactly on the pixel you point at in the display panel. The great advantage of the **Magnifier** window is that you can display a selection of the image with a completely different zooming factor:

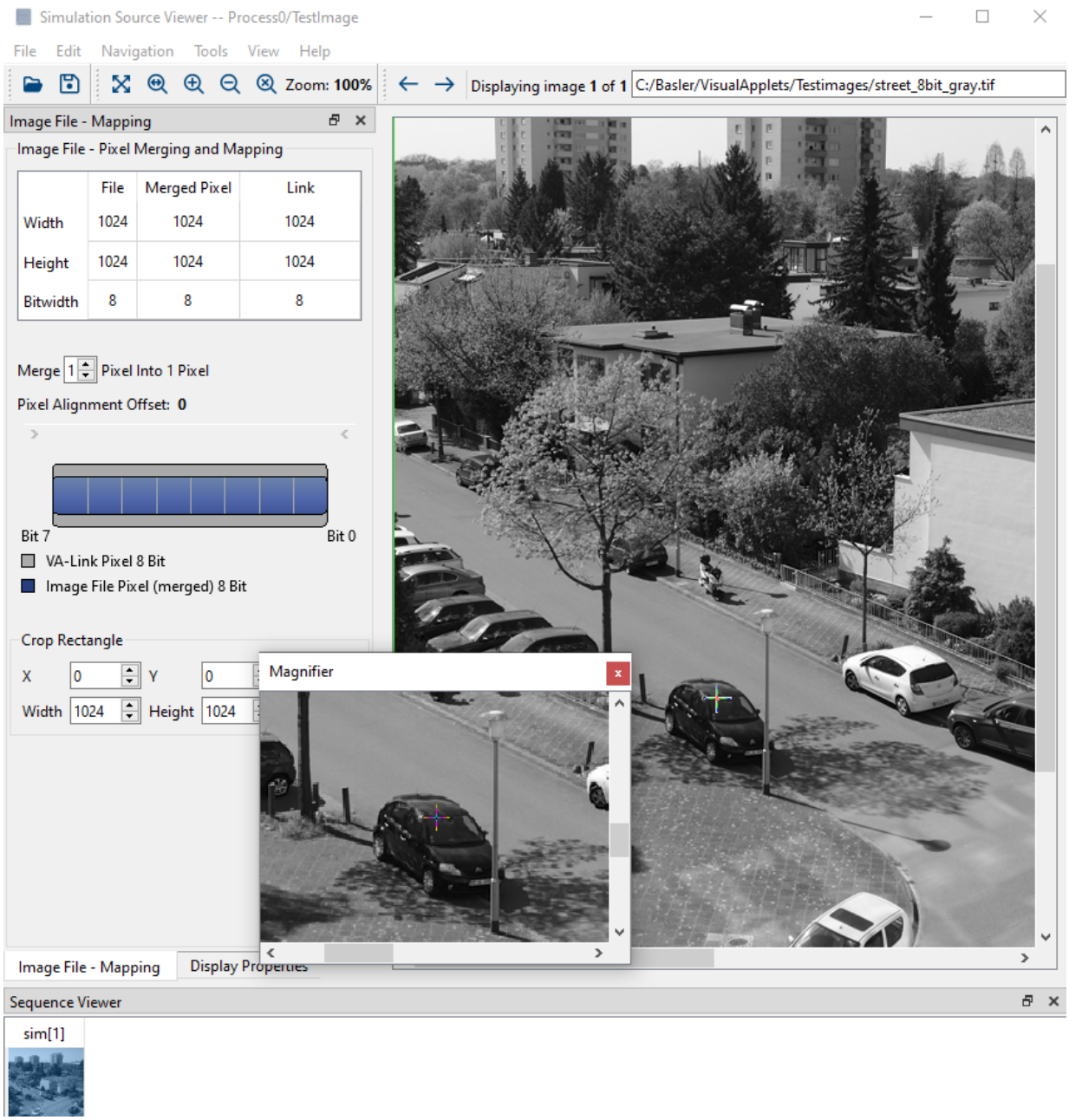



Figure 3.68. Zooming in the Magnifier

If you want to change the zooming factor of the **Magnifier**:

1. Activate the **Tools** -> **Magnifier** window.
2. Use **CTRL+MouseWheelUp** to zoom in and **CTRL+MouseWheelDown** to zoom out.

If you choose a very high zooming factor, a pixel grid is displayed for better orientation.

### 3.10.5.2. Loading Images into the Source

To load test images or test image sequences into your source,

1. Open the **Simulation Source Viewer** by double-clicking the source in your design.

2. From the main menu of the **Simulation Source Viewer**, select **File** -> **Open** or click the corresponding icon in the toolbar. In the following dialog, select the simulation input image file and click Open.



**Note**

You find some useful test images in the VisualApplets installation folder under `\Testimages`.



**Tip**

Alternatively, you can simply drag & drop image files either from your Windows Explorer into the viewer, or from your Windows Explorer onto the source in your diagram.

The input image is now displayed in the **Simulation Source Viewer**.

After closing the **Simulation Source Viewer**, you see a thumbnail of the image in the preview frame of the corresponding source:

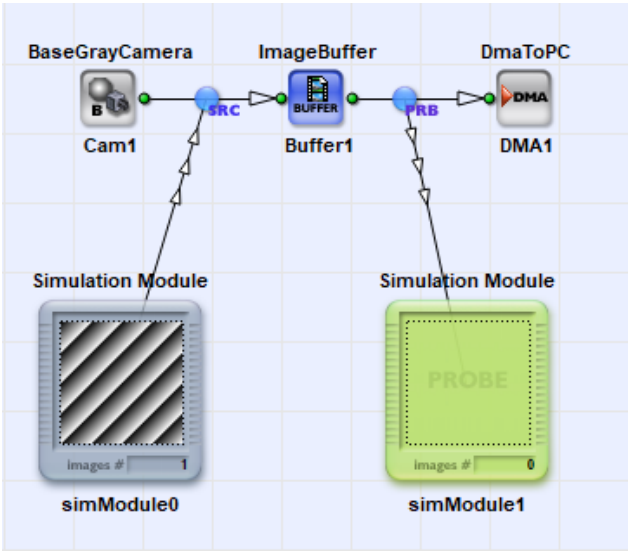



Figure 3.69. Thumbnail Display in Source

As soon as you connect a source to a link, the source takes over the parameter settings of the link (such as maximal image dimension, bit width, and color format). If you load an image that is bigger than the maximal image dimension of the link, a green rectangle is displayed on the image, enclosing the part of the image that will be used for simulation:

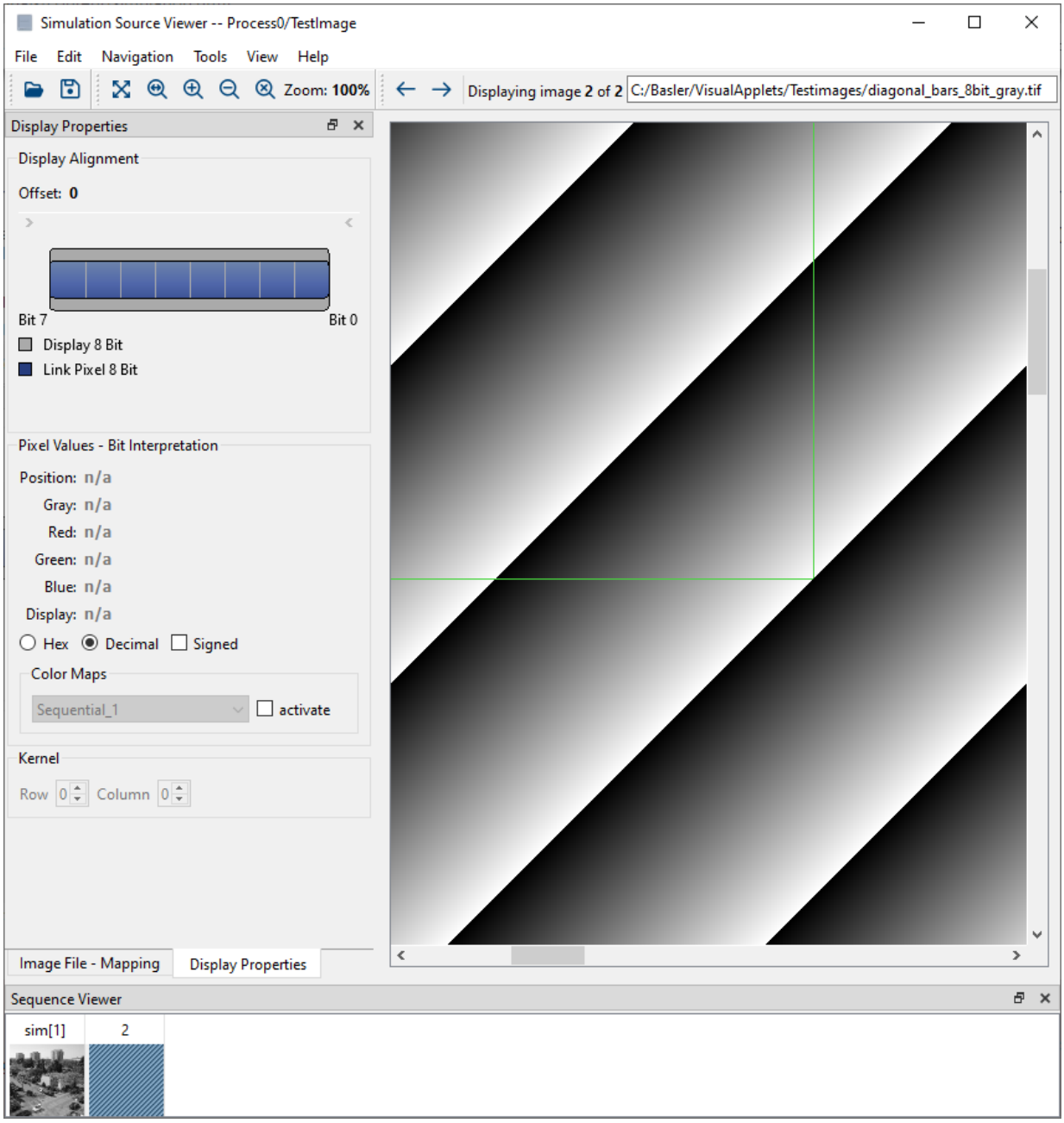



Figure 3.70. Highlighted Image Section Used for Simulation

**Convert your Images**

It is not possible to load a color picture on a one-channel gray link and vice versa. If your test image does not fit the properties of the link the source is connected to, you need to convert the image. Use any image processing program that offers the necessary functions to convert your test image. Use an image format supported by VisualApplets. For information on supported image formats, see Section 3.10.2, 'Supported Image Formats'.

### 3.10.5.3. Loading an Image Sequence into your Simulation Source

As soon as you load a new image to the source, the image shows up in the display panel of the **Simulation Source Viewer** and is added to the image sequence of your source. You can see all images

of your image sequence in the **Sequence Viewer**, which is located at the bottom of the **Simulation Source Viewer**.

Only the selected image displayed in the display channel is stored in the RAM of the computer. The other images of the sequence are loaded into the RAM when necessary. Thus, loading long image sequences onto a source has nearly no impact on RAM usage.

If you load an image sequence, you can change the order of images after load. In the **Sequence Viewer**, use drag & drop to position an image to where you want it to be in the sequence.

You can easily delete one or more images of a sequence:

1. Press **CTRL** and select the images you want to delete.

2. Press **DEL** or select from the menu **Edit** -> **Remove Selected**.

In the **Sequence Viewer**, **sim[x]** indicates the image that is being simulated during the next simulation step. You can reset **sim[x]** to the first image of the sequence in the *Simulation* dialog by clicking the **Reset** button. If you want to know more about simulation steps and reset, see Section 3.10.7, 'Setting the Number of Processing Cycles and Starting the Simulation'. The simulation order is not influenced by the image currently displayed in the display panel.

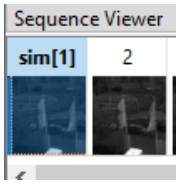



Figure 3.71. sim[x] Indicates the Image that Is Simulated in a Sequence

If you change the image properties by mapping (see Section 3.10.6.3, 'Image File Mapping '), the thumbnails are not refreshed automatically. By default, the **Sequence Viewer** displays the thumbnails of the original images.

You have two possibilities to adapt the thumbnail display. From the menu, select either

- **View** -> **Mapped Thumbs** to get a preview on how your images will look after applying the current mapping settings in a simulation, or

- **View** -> **Refresh** to have thumbnails displayed that reflect the current look of your images.

## 3.10.6. Optimizing your Image Input via Parameters

You can now optimize your image input via parameters. All parameters are displayed in the **Simulation Source Viewer**.

### 3.10.6.1. Pixel Values

On each pixel in the display panel, actually two crosshair cursors are displayed If you can’t see the two crosshair cursors, zoom in on the picture in the display panel:

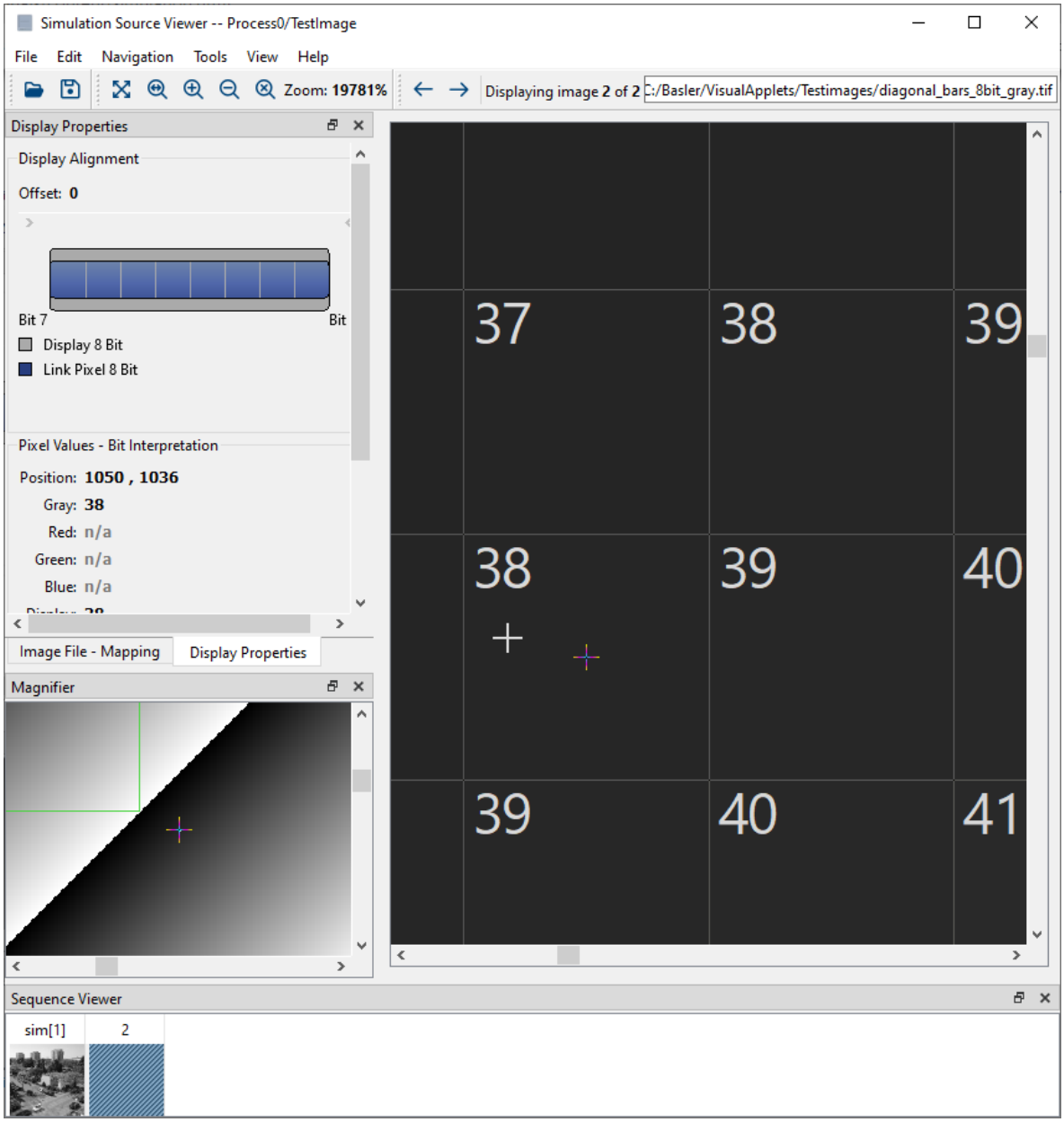



Figure 3.72. Crosshair Cursors in Display Window and Magnifier

The white/black crosshair cursor shows the position of your mouse cursor.

The colored one is positioned in the center of the pixel the mouse cursor points to. This crosshair cursor is also displayed in the **Magnifier** on exactly the same pixel.

In the settings panel tab **Display Properties** the corresponding pixel values are displayed:

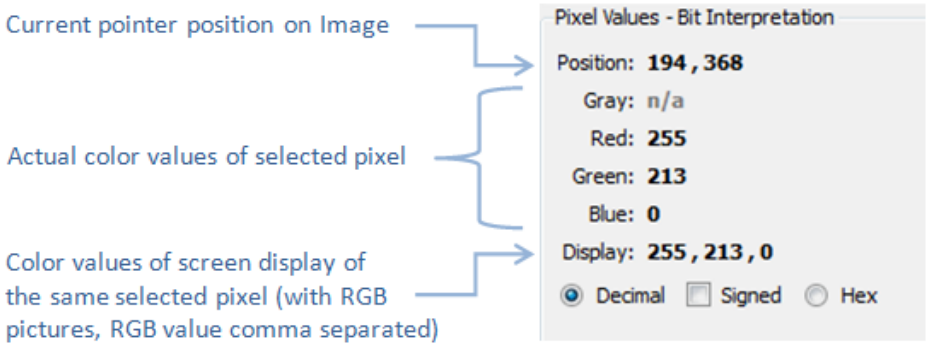



Figure 3.73. Pixel Values

You can choose whether you want to see the pixel values as decimal unsigned, decimal signed or hex figures.

Display shows the value(s) displayed on your screen. This value/these values might be different from the actual color values (or gray value) if a 16 bit or 48 bit image is loaded to your source, since on screen, only 8 bit can be displayed per color channel.

### 3.10.6.2. Image Dimension

In the **Simulation Source Viewer** tab **Image File – Mapping**, you find a table with the image dimensions of the current image (as it is loaded from file) on the left hand side and the image dimensions of the link on the right hand side:

Image File - Pixel Merging And Mapping

| | File | Merged Pixel | Link |
|---|---|---|---|
| Width | 1024 | 1024 | 1024 |
| Height | 1024 | 1024 | 1024 |
| Bitwidth | 8 | 8 | 8 |

Figure 3.74. Image Dimensions

The column **Merged Pixels** is described at Section 3.10.6.5, 'Pixel Spitting and Merging'.

If an image file has a dimension smaller than the dimension of the link, the image is transferred onto the link on the scale 1:1.

If the image file has a dimension that exceeds the dimension of the link, only part of the picture is used for simulation. You can see in the display panel in the green rectangle which picture part is going to be used for simulation. Also, the corresponding values are highlighted in yellow:

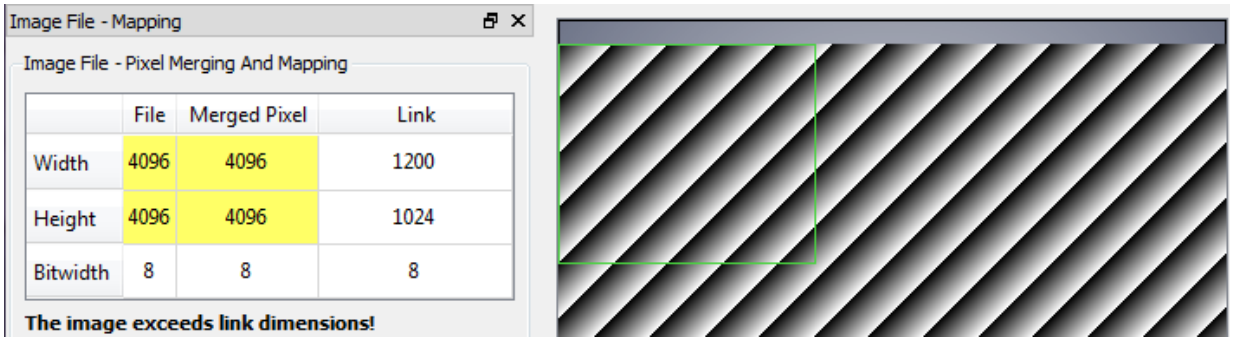



Figure 3.75. Exceeded Image Dimensions

**Tip**

If you cannot see the green rectangle in your display screen, set your viewing options to **View** -> **Fit to Window**.

### 3.10.6.3. Image File Mapping

In the **Simulation Source Viewer** tab **Image File – Mapping**, you also find information on the bit width of your test image (file) and on the bit width of the link the source is connected to:

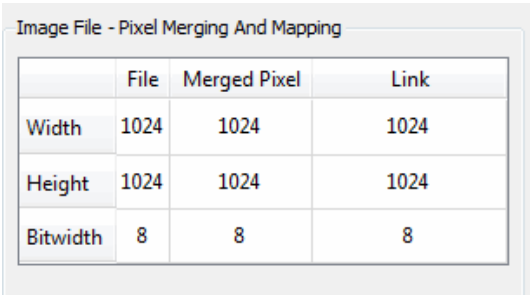



Figure 3.76. Bit Widths of Image and Link

The column **Merged Pixels** is described at Section 3.10.6.5, 'Pixel Spitting and Merging'.

If you want to load an image that has a bit width higher than that of the link, you can choose which bits of your image you want to use for simulation. You can make your selection by using a slider:

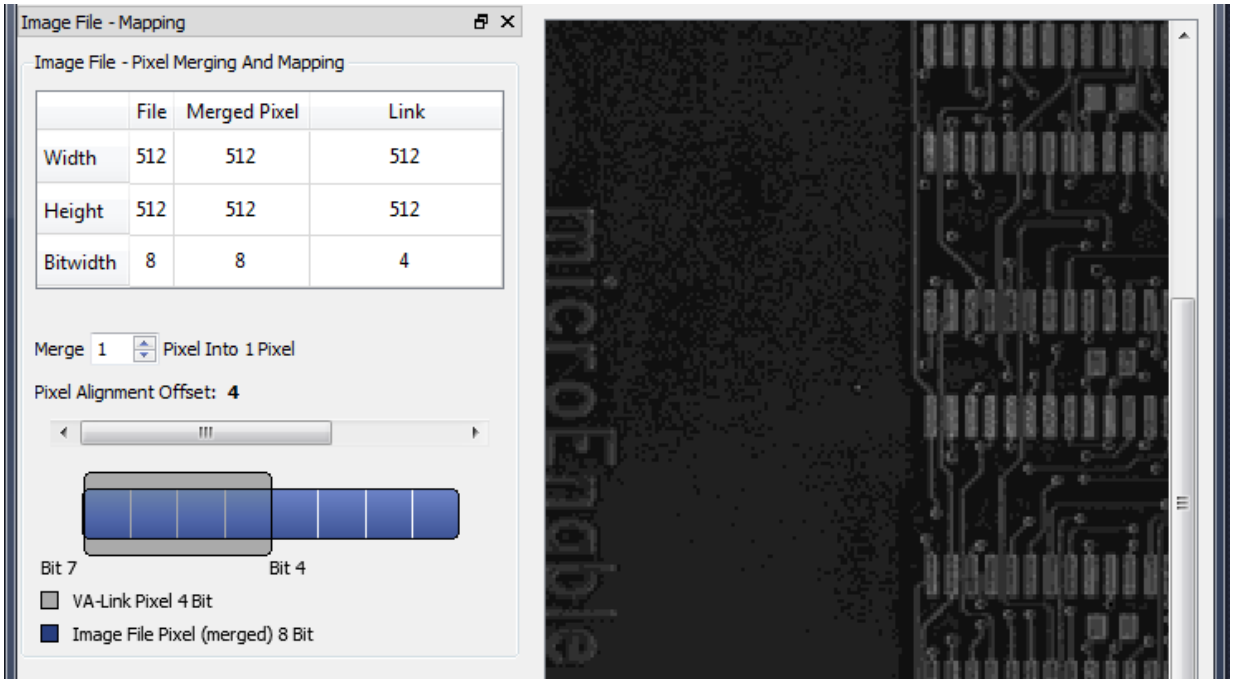



Figure 3.77. Defining Offset for Image Bits to Use

The bit width of the selection slider is the same as the bit width of the link. With the slider you can define a new position for the offset. The offset is the starting point of your bit selection on the bit width of your image. Based on these settings, the **Simulation Source Viewer** immediately calculates the altered image and displays it in the display panel. In the screen shot above, the 4 upper bits have been chosen out of the 8 bit of the 8-bit image. Thus, the test image used for the simulation will only have 16 instead of 256 gray-scale values.



**Visualization of Altered Test Image in the Simulation Source Viewer**

Since the display on PC monitors is based on 8-bit information per pixel, an image with only 16 gray-scale values cannot be displayed properly. Thus, for visualization on screen the 16 gray-scale values of the test image are mapped to the 256 gray-scale values of the 8-bit monitor display. This mapping doesn't change the number of shades of gray displayed (16), but enhances the contrast between them (0 = black, 15 = white).

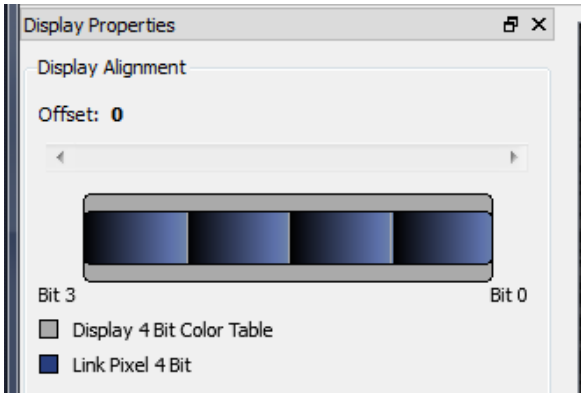



Figure 3.78. Display Properties for 4-bit Image

If you, on the contrary, load an image that has a smaller bit width than the link it is loaded on (i.e. the source module is connected to), you can use the same slider in tab **Image File – Mapping** to define an offset for the bits of the image on the bit width of the link. The remaining bits of the link are set to NULL.

The picture below shows an example where an 8-bit gray-scale image has been loaded onto a link with a bit width of 12 bit:

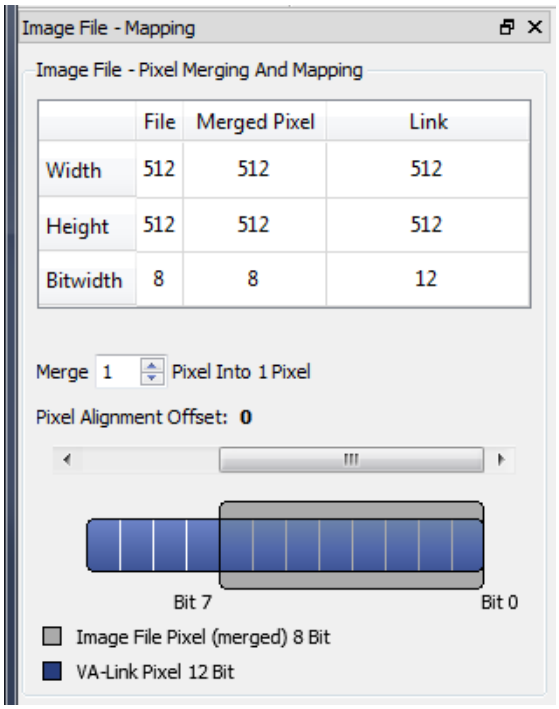



Figure 3.79. Defining Offset for Link Bits to Use

### 3.10.6.4. Display Alignment

You can define some settings for visualizing the altered test image in the **Simulation Source Viewer** via display alignment:

If the bit width of a link is higher than 8 bit, you have to define which 8 bits out of the link bit width you want to have displayed in the display panel. Since the display on PC monitors is limited to 8 bit per color channel, a higher bit width cannot be displayed.

In the **Simulation Source Viewer** tab **Display properties**, you can define which 8 bits out of the bits of a link you want to have displayed in the display panel. Use the slider to choose the offset for the displayable 8 bits. This display alignment setting has no influence on the simulation.

In the example below, the link has a bit width of 12 bit. With the slider, you can decide which bits out of the 12 you want to have displayed.

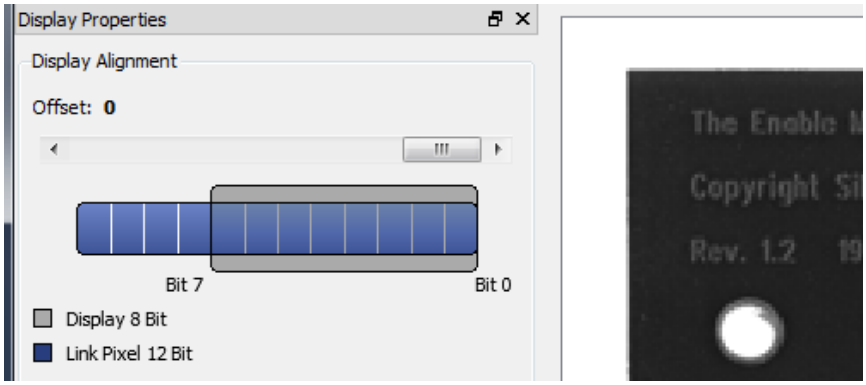



Figure 3.80. Display Alignment

### 3.10.6.5. Pixel Spitting and Merging

The BMP file format allows a maximum of 8 bit per color channel of a pixel. The TIFF format allows a maximum of 16 bit per color channel. A higher bit width cannot be simply saved in standard image file formats.

VisualApplets offers the possibility of pixel splitting to get around these limitations.

To store an image with a high pixel bit width in a standard image file format, each pixel is split into multiple image file pixels. Thus, it is possible to store up to 64 bit per color channel in BMP or TIFF file format, e.g., in 8 x 8-bit pixels. These images can be viewed in regular image file viewers or image processing programs. Of course, the VisualApplets generated pixel-split-images will be displayed horizontally expanded. Thus, in VisualApplets, pixel splitting is a mere storing option for images with a high bit width (color depth).

To load an image with high bit width that has been stored as a BMP or TIFF file into your simulation source, you have to merge the adjacent pixels that share the color information for one original pixel back to one pixel. Thus, simulation source viewers can merge the pixel-split-images for correct mapping. The splitting can be done in the **Simulation Probe Viewer** (see Section 3.10.8, 'Evaluating the Simulation Results'). You can define the settings for pixel merge in the **Simulation Source Viewer** tab **Image File – Mapping**:

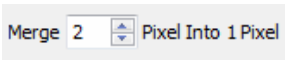



Figure 3.81. Pixel Merge

In the following example, a 2048x1024 BMP file with a bit width (color depth) of 8 bit has been loaded into the source. It is to be interpreted as a 16 bit 1024x1024 image.

If you enter 2 in the Merge N pixel into 1 pixel field, the bit width of the image in your source changes from 8 bit to 16 bit per color channel, whereas the row length of the image in your source will be only half as long, thus changing from 2048 to 1024 pixel.

Before merge:

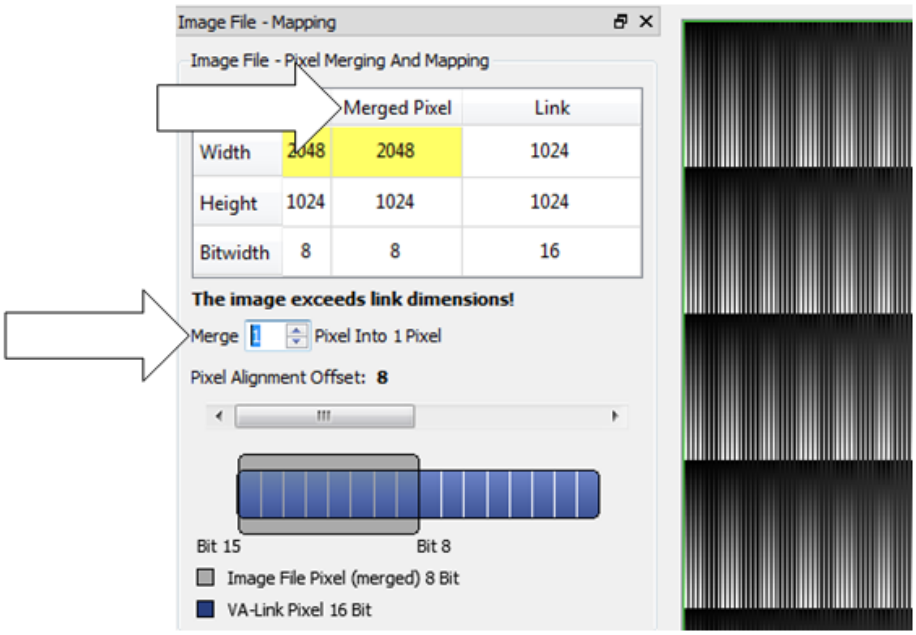



Figure 3.82. Merging Factor = 1, Image Properties Do Not Fit Link Properties

After merge:

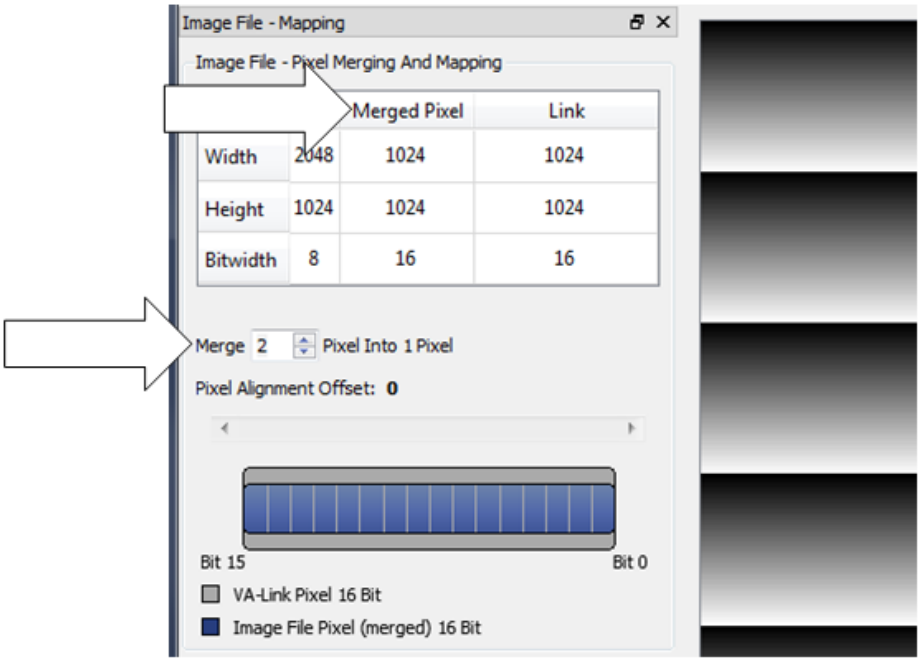



Figure 3.83. Merging Factor = 2, Properties of Merged Image Fit Link Properties

## 3.10.7. Setting the Number of Processing Cycles and Starting the Simulation

After you have loaded a test image or test image sequence to your source and defined all settings for the image, you can start the simulation.

**Use Simulation Probes**

Make sure you inserted (and connected) simulation probes on all positions where you want to check image processing results (see Section 3.10.4, ' Inserting Sources and Probes into your Design').

1. From the main menu, select **Analysis** -> **Start Simulation**, or click the toolbar icon **Start Simulation**. Now, VisualApplets automatically triggers a Design Rules Check 1.

   The **Simulation** window opens up:

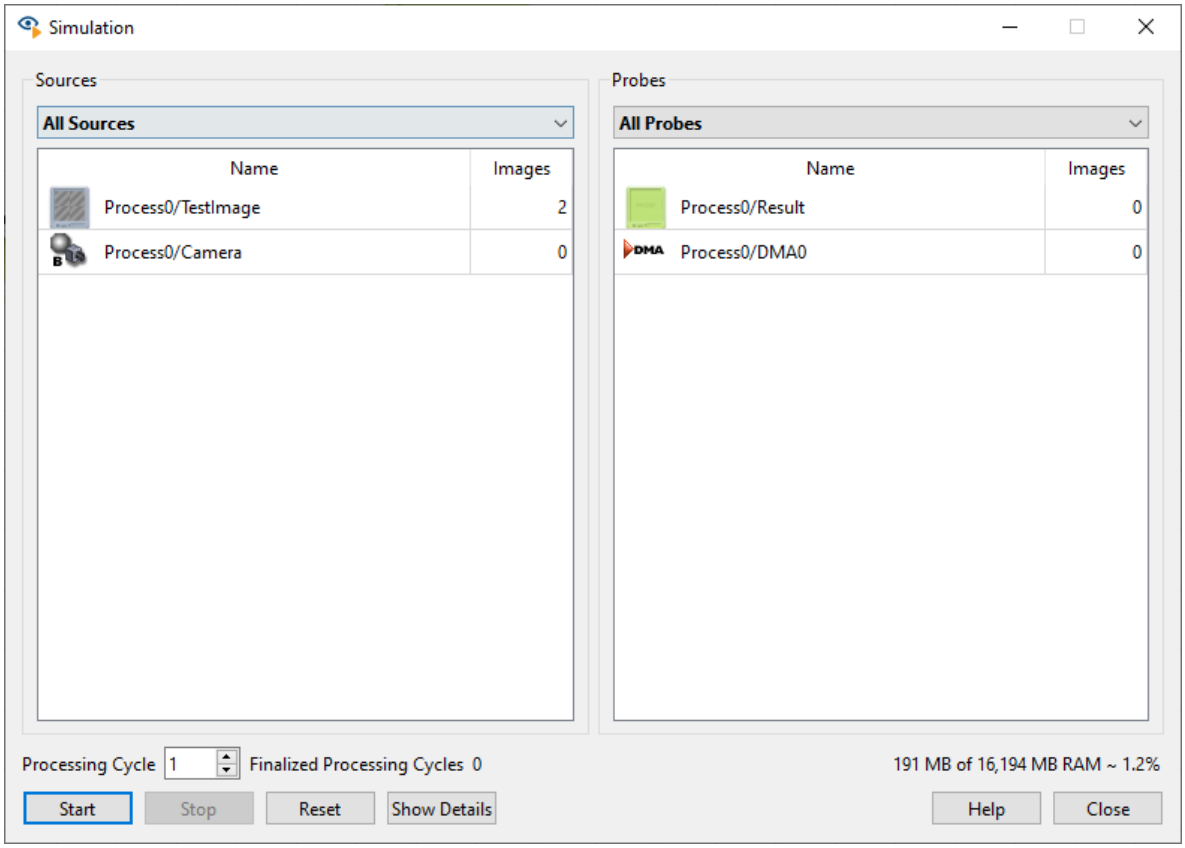



Figure 3.84. Simulation Window

In the upper pane, all simulation sources and simulation probes currently active in the design are listed. The list serves only information purposes.

In the lower pane of the window, you see the results of Design Rules Check 1.

You can change the display of the list via a drop-down list box:

- Simulation Sources and Simulation Probes display only the sources and probes you have set for simulation.
- **All Sources** and **All Probes** display all data sources and data destinations within your design.

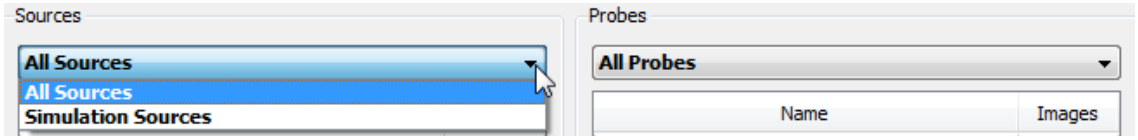



Figure 3.85. Changing Source and Probe Display

### Influence of Errors and cautions

A simulation can only be started if no error was detected during Design Rules Check 1. Warnings created by Design Rules Check 1 do not prevent starting the simulation.

**Note**

Single operators cannot be selected for simulation. A simulation will always run through all operators of a design.

If the list in the main simulation window shows a simulation source or probe marked by a yellow warning triangle, this source/probe is not connected to a link of your design and will be ignored during simulation.

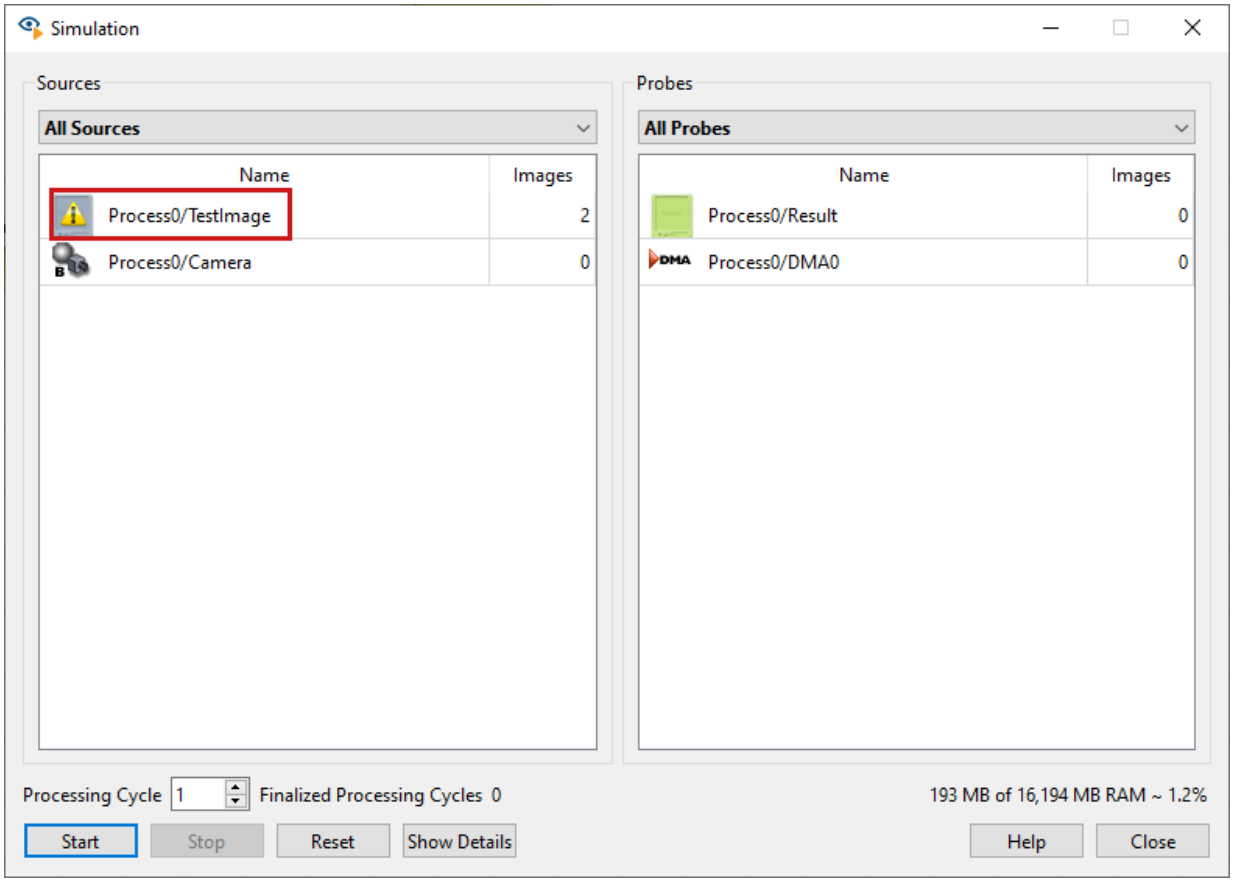



Figure 3.86. Non-connected Simulation Modules

**Note**

Camera operators do not emit data during simulation. One needs to connect a simulation source to the output link to provide data for the simulation.

In the Processing Cycle field, you can specify how many processing cycles you want the program to carry out.

You can also configure the simulation and the processing cycles via the **System Settings** menu: **Settings** -> **System Settings** -> Simulation:

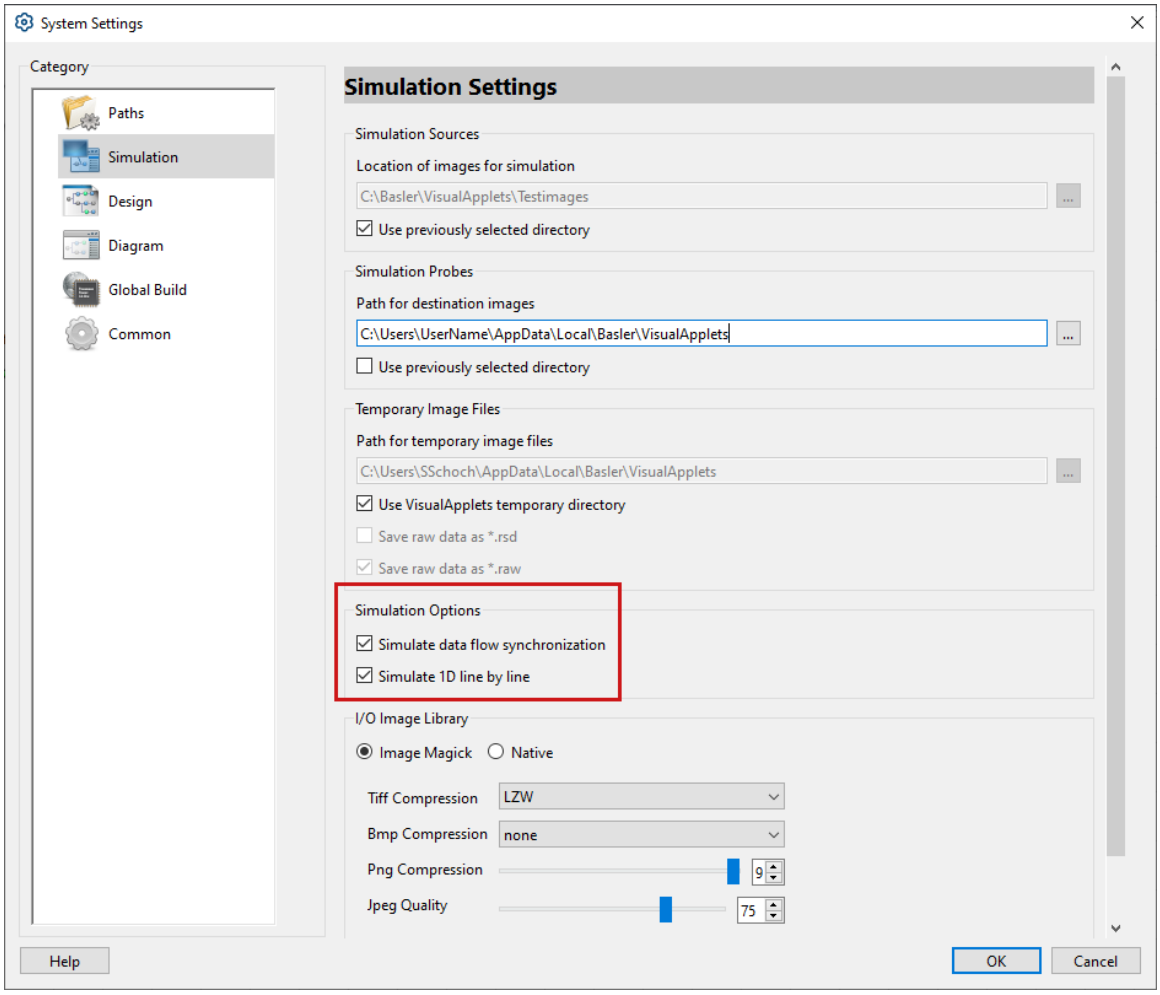



Figure 3.87. Simulation Settings

**Simulate data flow synchronization** cleared: Each source in your design, e.g. simulation sources and operators like *CreateBlankImage*, or *CoefficientBuffer*, emits one data set per cycle. The cycle ends, if no more data is to be processed or transported. A simplified data management allows operators to divert from the behavior of the corresponding entity in hardware. This means, instead of storing incoming data inside the input ports, an operator might implement unrealistically large internal buffers.

**Simulate data flow synchronization** selected: The design as a whole is simulated once per cycle from source(s) to sink(s). The focus is on precise modeling of dependencies and accurate data management. Sources are permitted to provide one or more data sets per cycle. The simulation might prevent sources from emitting data if previously generated data is still to be processed. Therefore, operators leave incoming data inside their input ports until data processing requires it to be collected. As long as valid data is stored in the incoming port, upstream operators are prevented from producing data. This causes congestions and corresponds to the concept of inhibits in hardware and allows for the detection of deadlocks.

**Simulate 1D line by line**: 1D data can be processed by aggregating several lines to one 2D frame and simulates it like any other 2D data. This is fast and robust but limits the simulation to exactly one simulation cycle. If you select **Simulate 1D line by line**, each line is processed individually and thus allows multi-step simulations, e.g., to deal with 1D loops. This feature relies on data flow synchronization, this means you can only select this option, if **Simulate data flow synchronization** is selected.



**Behavior of Simulation Probes During Simulate 1D line by line**

For simulation probes connected to 1D links, incoming line data is appended to the last frame in the simulation probe. After each processing cycle, the respective lines are

finalized and a new frame is started. While the simulation dialog is open, the **Sequence Viewer** is locked.

After selecting the desired amount of Processing Cycles in the **Simulation** view, the simulation can be run by clicking the **Start** button. As the internal status of all operations is maintained, consecutive runs can be launched from the same window, as long as the simulation dialog is kept open. Once the dialog is closed, all operators are reset to their initialization state.

Example: A given *AppendImage* operator expects three input images for concatenation. A connected source provides one image per processing cycle. The simulation is run for two cycles, a simulation probe connected to the output link of *AppendImage* does not show any data yet, since *AppendImage* still lacks one additional image to start the processing.

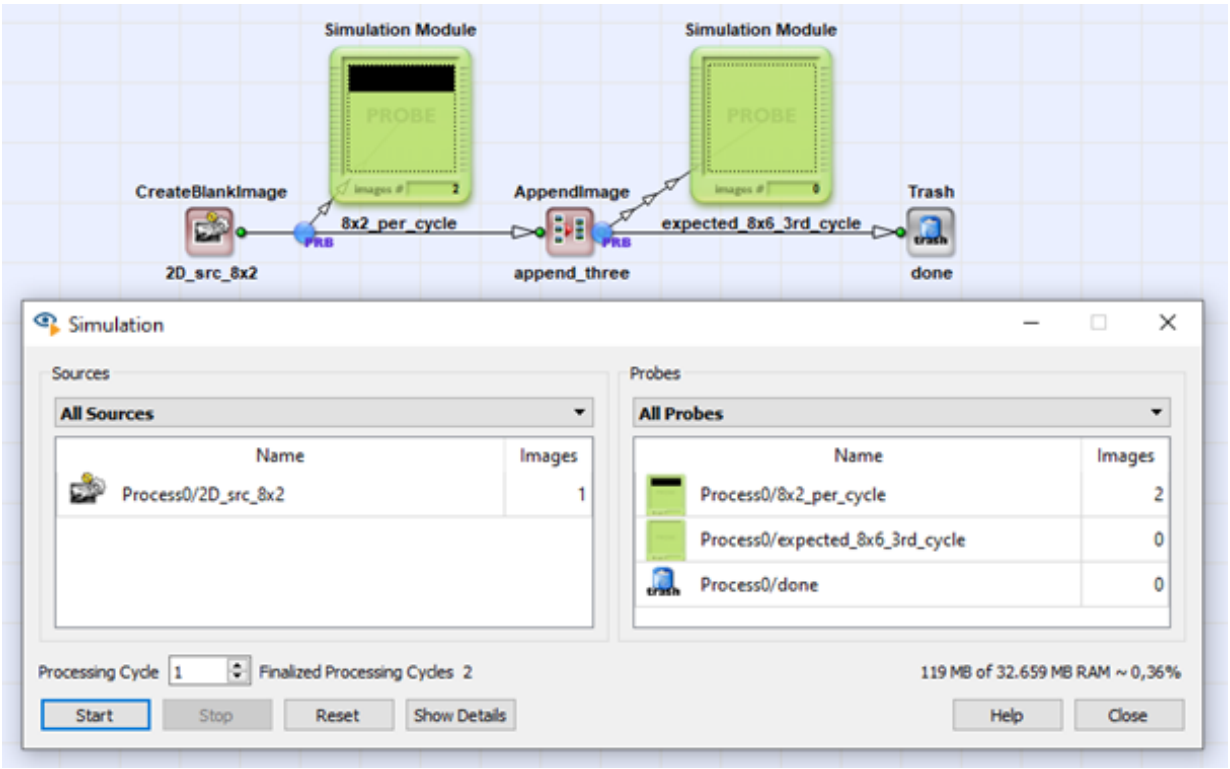



Figure 3.88. Second Simulation Step

While the simulation dialog is still open, one can perform an additional (third) simulation cycle which in turn provides data at the connected simulation probe.

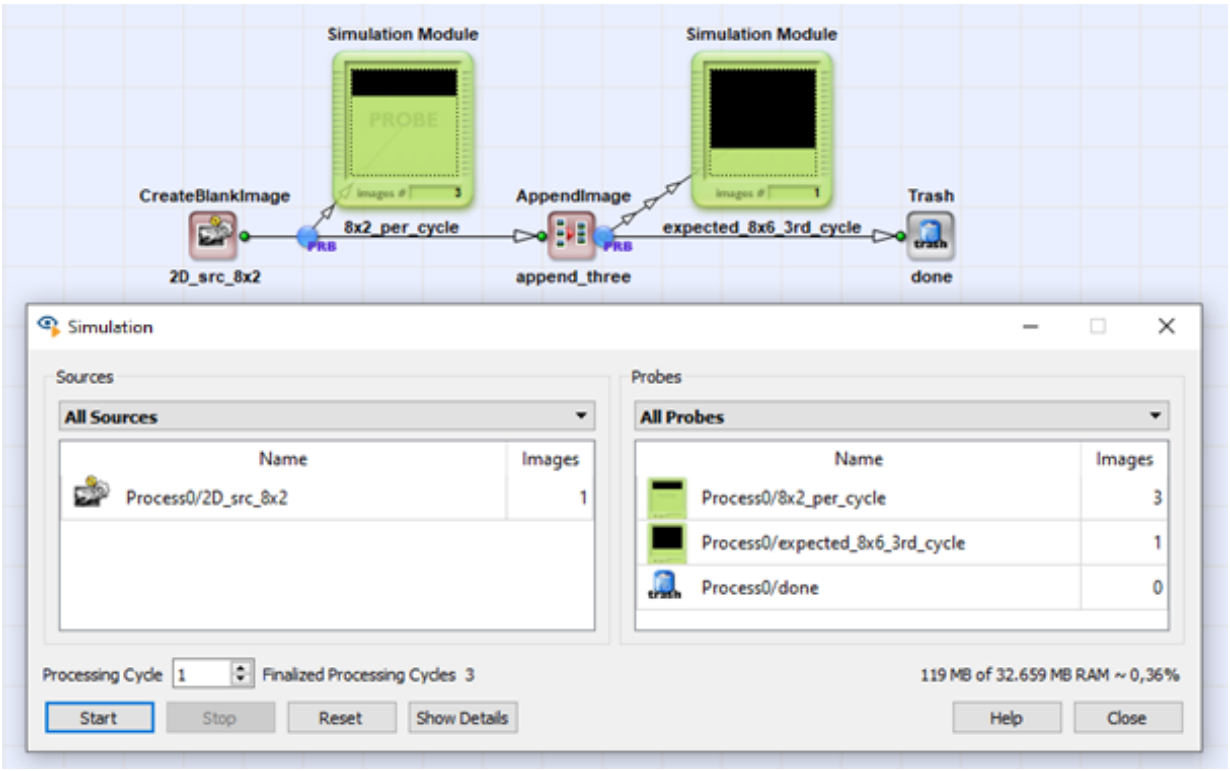



Figure 3.89. Third Simulation Step

If the dialog was closed and reopened in the meantime, the additional simulation cycle would be executed after initializing the design and therefore would provide the first of the three required images for *AppendImage*.

**Processing Order of Images within a Source Module**

In a new processing cycle, always the next image of the image sequence loaded to the source is processed. When all images of the sequence have been processed, the first image of the sequence is processed again.

If only one image is loaded to the source, this image is processed again and again with each processing cycle. For example, if you specify four processing cycles, the one image of the source is being processed up to four times.

Depending on your design, the number of images fed into the design might differ from the number of images you get as an output, for example when you use the operators *SplitImage* or *RemoveImage*. Thus, for one processing cycle, you might get more or less images as result(s) in a probe than you have source modules in your design. See also the example Section 9.2, ' Multiple DMA Channel Designs ' in the VisualApplets tutorial.

To clear the simulation and reset it to the start-up condition, use the **Reset** button.

A reset produces the following effects:

- The image sequence in all simulation sources is reset, that is, a new simulation will start with the first image of the sequence again.
- The content of all probes is deleted.
- All operators are reset to their start-up condition (e.g., image counters, uncollected data in input ports).

Now, you are ready to start the simulation of data processing as defined in your design.

2. Click **Start** to start the simulation.

The current simulation status is displayed in a progress bar at the bottom of the *Simulation* view. Warnings and errors are displayed in the *Log* panel, which you can open or close by clicking the **Show Details**/**Hide Details** button. As soon as warnings or errors occur, the *Log* panel is displayed automatically.



**Tip**

If you want to, you can keep the viewer windows of your simulation modules open during simulation. However, this will slow down simulation.

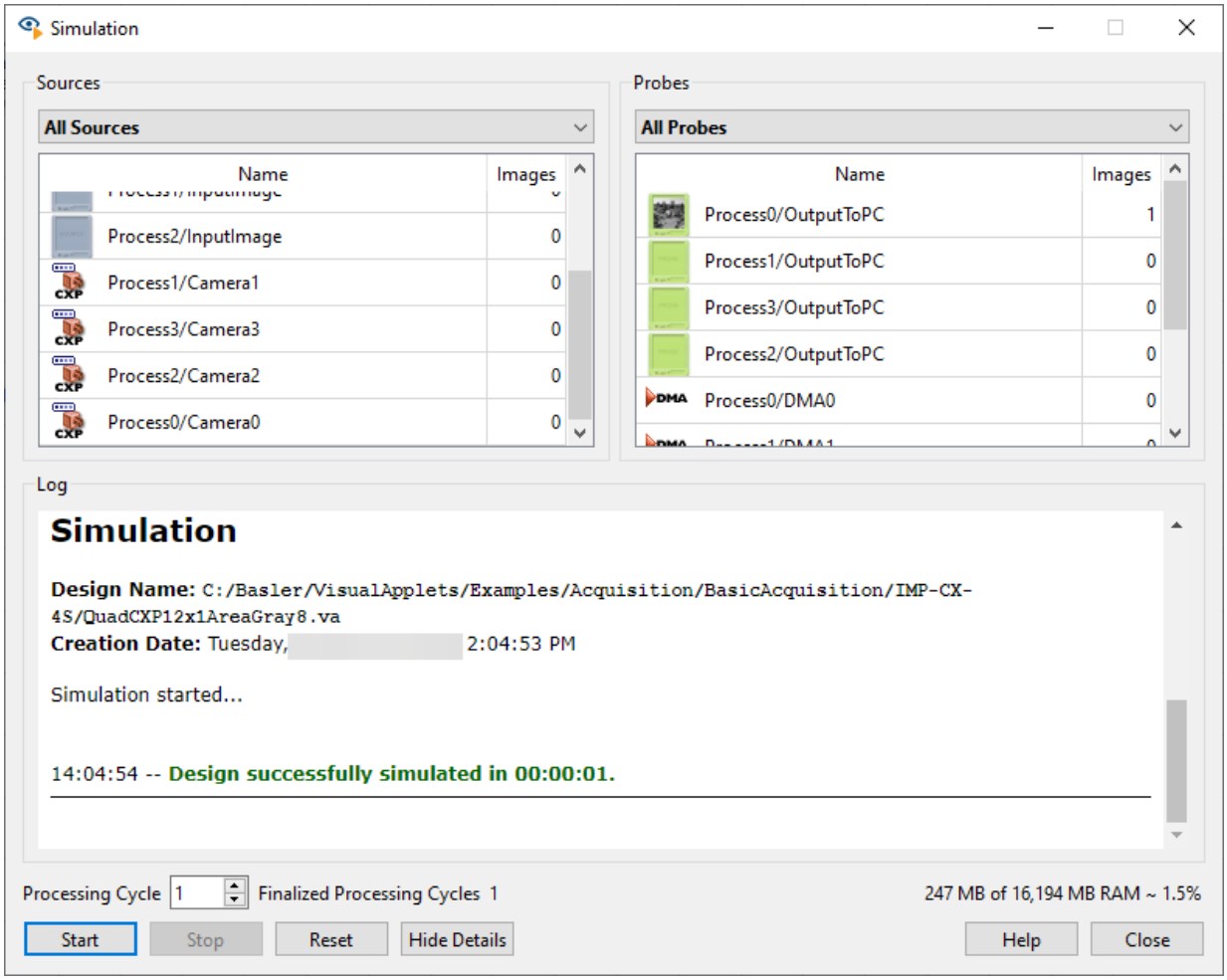



Figure 3.90. Successful Simulation

While a simulation is in progress, the simulation probe modules are filled with the resulting images. Thumbnails of the simulation results are displayed in the preview image frames. After opening the simulation viewer window of a simulation probe, the result(s) will be displayed in full size.



**Automatic Simulation Reset**

The simulation will automatically reset when you change your design. On reset, all simulation probes are cleared, too.

## 3.10.8. Evaluating the Simulation Results

After the simulation, you find the simulation results in the probes.

**Some Images Are Not Displayed in Probes**

When simulation probes contain very large images, VisualApplets may fail to display these images correctly due to memory limitations. In that case, a gray image (i.e. all pixels have the value 205 (0xCD)) is shown. If this happens, use smaller images for your simulation.

Double-click a probe in your design to open the **Simulation Probe Viewer**.

It looks similar to a **Simulation Source Viewer** (see Section 3.10.5, 'Loading the Image File(s) to your Simulation Source(s) '): The settings channel is located at the left hand side, the display channel on the right hand side, and the **Sequence Viewer** at the bottom of the **Simulation Probe Viewer** window.

In the **Simulation Probe Viewer**, you get the simulation results displayed for a first evaluation. You can alter the display options by pixel alignment, and save the images to file. For saving, there are several options available, like, e.g., pixel splitting.

You can zoom in and out on the image displayed in the display channel by either using the corresponding icons in the toolbar, or by using **CTRL+MouseWheelUp** to zoom in and **CTRL+MouseWheelDown** to zoom out.

If you choose a very high zooming factor, a pixel grid is displayed for better orientation and the corresponding pixel values are textually represented.

### 3.10.8.1. Pixel Values

Like in the **Simulation Source Viewer**, on each pixel in the display panel two crosshair cursors are displayed. If you can't see the two crosshair cursors, zoom in on the picture in the display panel. The white/black crosshair cursor shows the current position of the cursor of your mouse, whereas the colored one is positioned in the center of the pixel the mouse cursor points on. This crosshair cursor is also displayed in the *Magnifier* on exactly the same pixel.

In the settings panel, the corresponding pixel values are displayed:

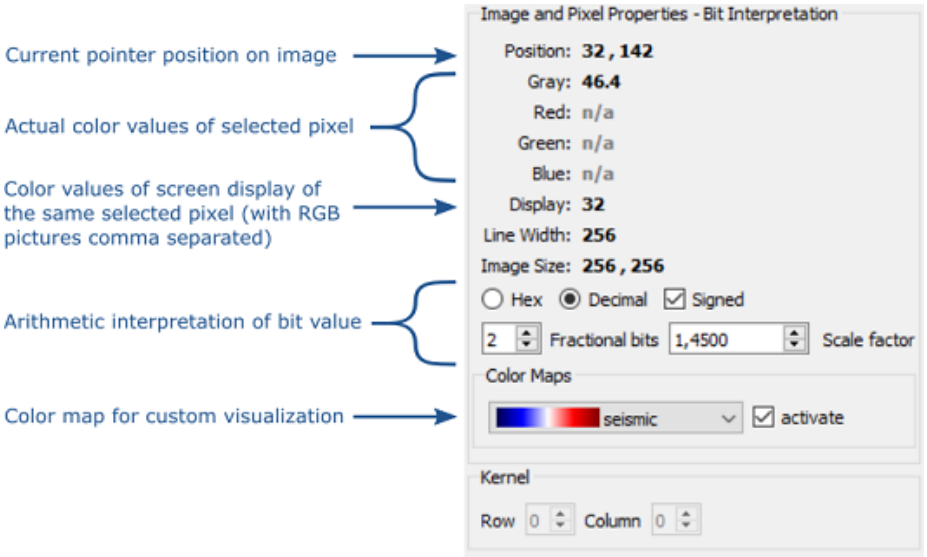



Figure 3.91. Pixel Values Probe

You have different options to adjust the interpretation of color values for the selected pixel:

- You can choose whether you want to see them as hexadecimal numbers or as signed or unsigned decimal numbers.
- You can set the number of fractional bits for an interpretation as fixed-point numbers.

- Additionally, you can define a scaling factor which is multiplied to the values. This is useful for performing linear conversions like radiant to degree.

Display shows the value(s) displayed on your screen. This value/these values might be different from the actual color values (or gray-shade value) if a 16 bit or 48 bit image is in your probe, since on screen, only 8 bit can be displayed per color channel.

You may select a color map which performs tone mapping between the values in *Display* and the shown color in the image window. This is especially useful when dealing with signed image data or when the contained image characteristics are poorly visible in a grey-scale representation.

If the simulation probe is connected to a link which transports kernels, you can select the individual kernel images in the kernel *Row* and kernel *Column* spin boxes in the *Kernel* panel.

### 3.10.8.2. Image Sequence

Once an image is simulated, it is instantaneously displayed in the **Simulation Probe Viewer** and added to the **Sequence Viewer**.

You cannot change the order of images in the **Simulation Probe Viewer**.

However, you can delete one or more images of a sequence:

1. Press **CTRL** and select the images you want to delete.
2. Press **DEL** or select from the menu **Edit** -> **Remove Selected**.

If you change the image properties by mapping (for example, when you save 16-bit images in BMP files), the thumbnails are not refreshed automatically. By default, the **Sequence Viewer** displays the thumbnails of the original simulation results.

### 3.10.8.3. Varying Row Length

VisualApplets is not limited to processing and displaying rectangular images with homogeneous row length. It can also display images with rows that are of different length.

The image size for such images is calculated by VisualApplets as follows:



**Calculating Image Size for Images with Inhomogeneous Row Lengths**

Image size = number of pixels of longest row * number of rows

Rows with row length NULL are counted as well. The undefined areas of the image are represented in the display by a blue-to-cyan color gradient:

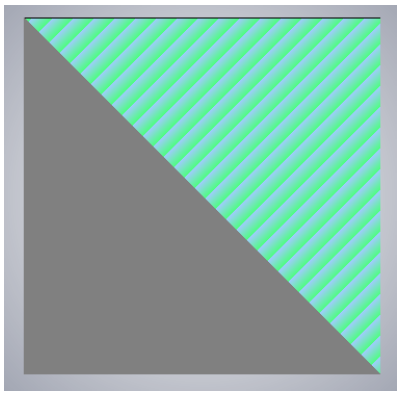


Figure 3.92. Display of Undefined Image Areas

Empty images are displayed by the symbol **Empty Image** and cannot be saved.

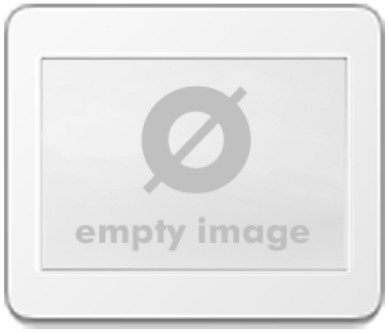



Figure 3.93. Empty Image Symbol

### 3.10.8.4. Display Alignment

The display panel of the **Simulation Probe Viewer** offers two ways of displaying the same image:

- If you activate *Link View*, the image is displayed as it looked like when it was passed to the probe by the link.
- If you activate *File View*, the image is displayed as it will look when you save it with the current settings to file.

Since monitors always use 8 bit per color channel, images with higher or lower bit width cannot be displayed without further ado.

If a simulation result has a bit width higher than 8 bit, you can use the offset slider to select the 8 bit you want to have displayed in the display panel:

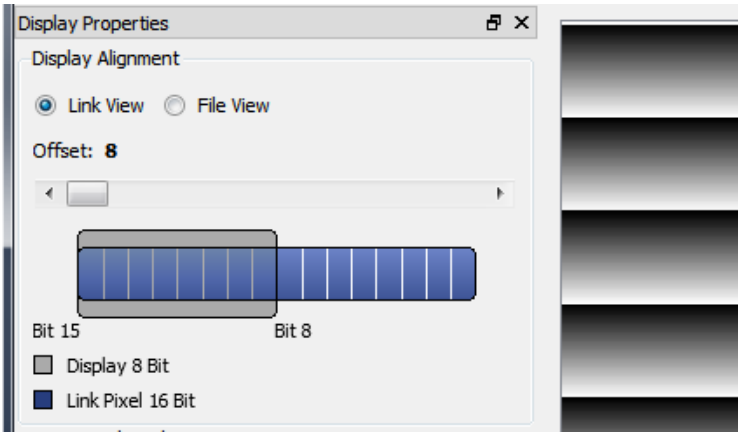



Figure 3.94. Link View

In this example, the upper 8 bits of a 16-bit gray scale are chosen for display.

If the bit width of the image is smaller than 8 bit, the gray-scale values of the test image are automatically mapped to the 256 gray-scale values of the 8-bit monitor display. This mapping doesn't change the number of shades of gray displayed, but enhances the contrast between them. Without this mapping, the human eye might not be able to see an image at all. Take, for example, 1-bit images: If the values 0 and 1 are used for monitor display without mapping, both are interpreted as black by the human eye. In this case, the automatic bit width mapping of VisualApplets interprets 0 as 0 (black) and 1 as 255 (white).

### 3.10.9. Line Profile

The **Line Profile** view shows the individual color values or gray values of a line. You can open the **Line Profile** via the menu **Tools**.

There are two output areas:

- **Overview**: The individual color values or gray values of the entire line are displayed.
- **Detail**: Enlarges an area marked in the **Overview** view. The area that is currently displayed in the **Detail** view is highlighted in the **Overview** view with a white background. You can move the area displayed in the **Detail** view with the scroll bar under the **Detail** view.

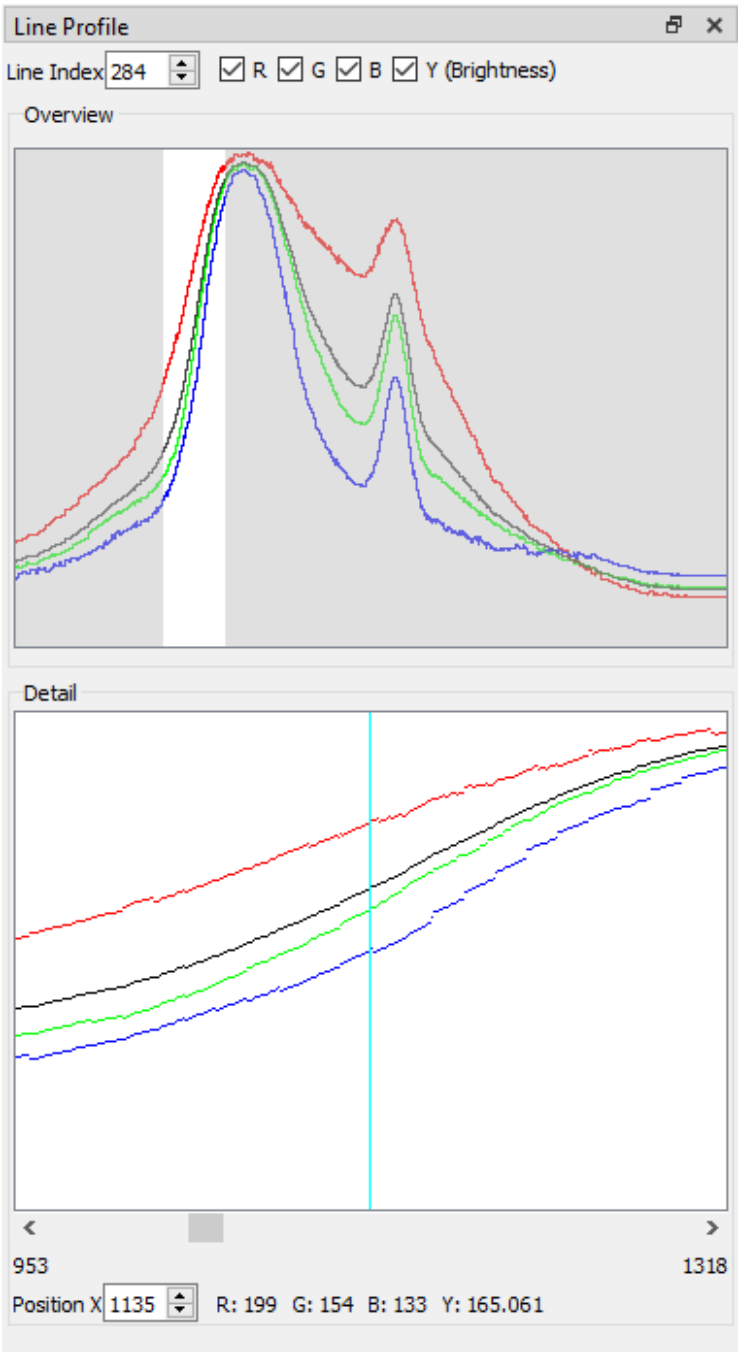



Figure 3.95. Line Profile View

With the spin box **Line Index**, you can select a certain line in the image and the line is marked in the **Display** panel.

For color images, you can switch the individual color channels on or off with the checkboxes **R**, **G**, **B**, and **Y**. **Y** stands for brightness. In addition, you can select a specific pixel position in the line via the spin box **Position X** or with the cyan marker. To use the cyan marker, just move the mouse into the **Detail** view. The values are then displayed in **R**, **G**, **B**, and **Y**.

## 3.10.10. Line Histogram

The **Line Histogram** view is a representation of the distribution of color values or gray values only of the selected line in an image. You can open the **Line Histogram** via the menu **Tools**.

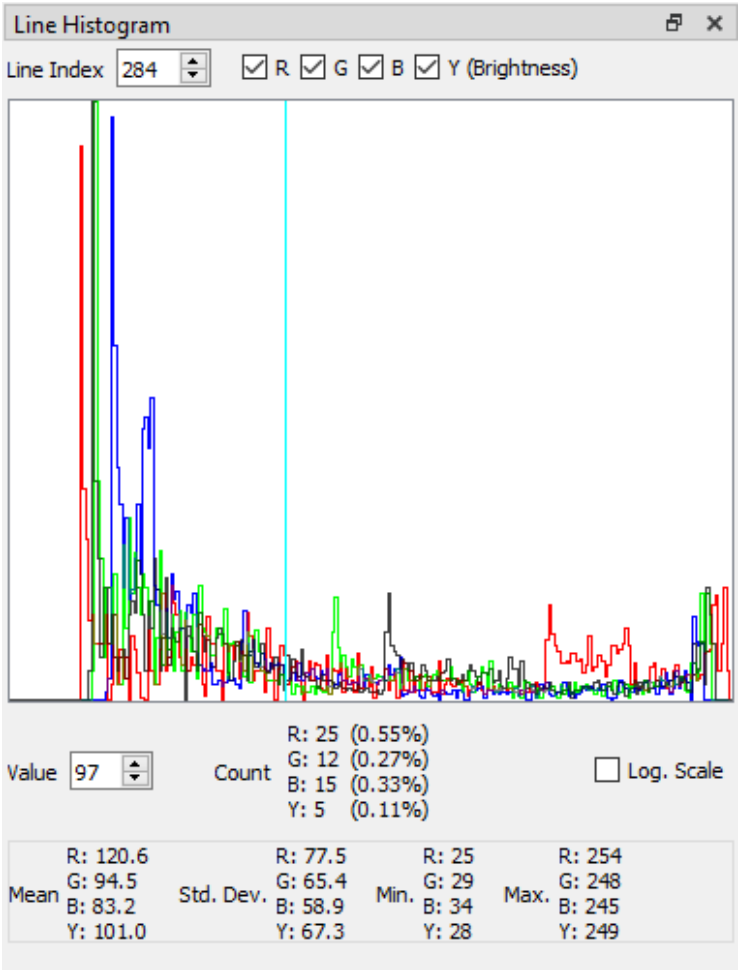



Figure 3.96. Line Histogram View

The histogram view is scaled so it always spans the full height of the diagram.

With the spin box **Line Index**, you can select a certain line in the image. As in the **Line Profile** view, the line is then marked in the **Display** panel.

For color images, you can switch the individual color channels on or off with the checkboxes **R**, **G**, **B**, and **Y**. **Y** stands for brightness. In addition, you can select a specific color value via the spin box **Value** or with the cyan marker. To use the cyan marker, set the mouse cursor into the output view.

For larger color depths, for example 16-bit images, the spin box **Bin** is displayed instead of the spin box **Value** and the bin ranges are displayed next to it: 

For a selected **Value** or **Bin** number, the number of pixels with values in the corresponding range are shown in **Count**. You can manually enter the number in **Value** or **Bin** or select the number by pointing the mouse to the diagram.

For color images, the output is split into the basic color values RGB and the brightness value Y.

### 3.10.10.1. Displaying Statistical Data

The **Line Histogram** view also displays the following statistical values, which are calculated for the selected line:

- **Mean** value
- **Standard Deviation (St. Dev.)**
- **Minimal (Min.)** value
- **Maximum (Max.)** value

## 3.10.11. Image Histogram

The **Image Histogram** view is a representation of the distribution of color values or gray values in an image. You can open the **Image Histogram** via the menu **Tools**.

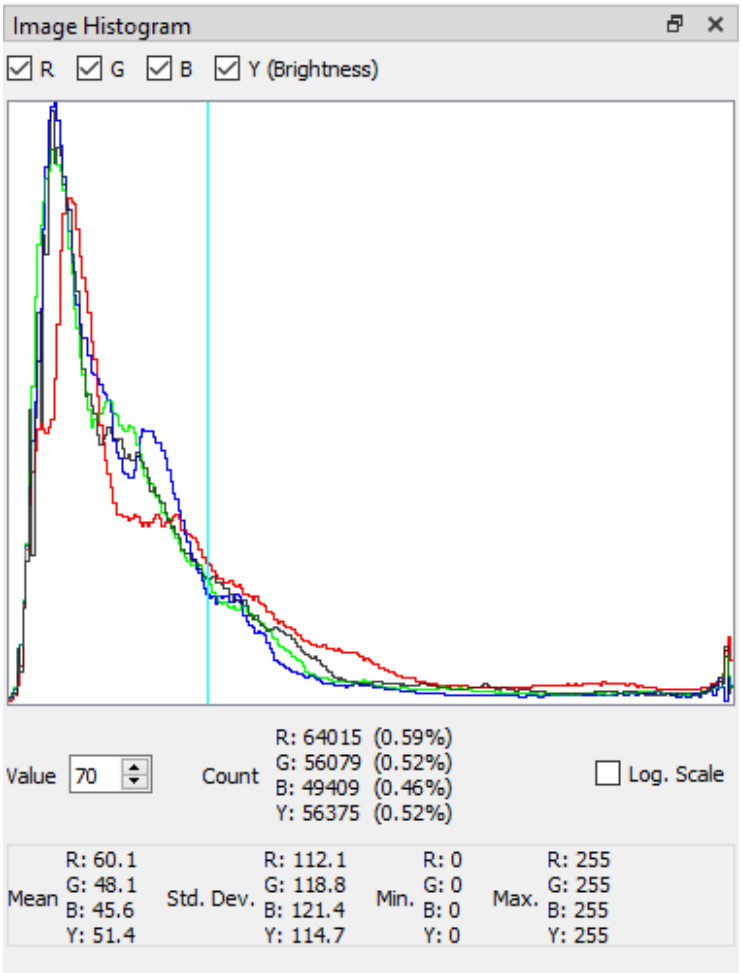



Figure 3.97. Image Histogram View

The histogram view is scaled so it always spans the full height of the diagram.

For color images, you can switch the individual color channels on or off with the checkboxes **R**, **G**, **B**, and **Y**. **Y** stands for brightness. In addition, you can select a specific color value via the spin box **Value** or with the cyan marker. To use the cyan marker, set the mouse cursor into the output view.

For larger color depths, for example 16-bit images, the spin box **Bin** is displayed instead of the spin box **Value** and the bin ranges are displayed next to it: 


For a selected **Value** or **Bin** number, the number of pixels with values in the corresponding range are shown in **Count**. You can manually enter the number in **Value** or **Bin** or select the number by pointing the mouse to the diagram.

For color images, the output is split into the basic color values RGB and the brightness value Y.

### 3.10.11.1. Displaying Statistical Data

The **Image Histogram** view also displays the following statistical values, which are calculated for the selected image:

- **Mean** value
- **Standard Deviation (St. Dev.)**
- **Minimal (Min.)** value
- **Maximum (Max.)** value

## 3.10.12. Saving Simulation Results

You can save your simulation results in the image file formats TIFF/TIF, BMP, JPEG/JPG, PNG, GIF, and PSD. The TIFF format offers native support of 8 and 16 bit per color channel, BMP offers only 8 bit per channel.

Before you actually save your image as TIFF/TIF or BMP file, you should define all saving settings in the *Save Options* dialog.

1. To open the dialog, from the main menu of the **Simulation Probe Viewer** select **File** -> **Save Options**. As a result, the Save Options panel opens up on the right hand side of the **Simulation Probe Viewer**.

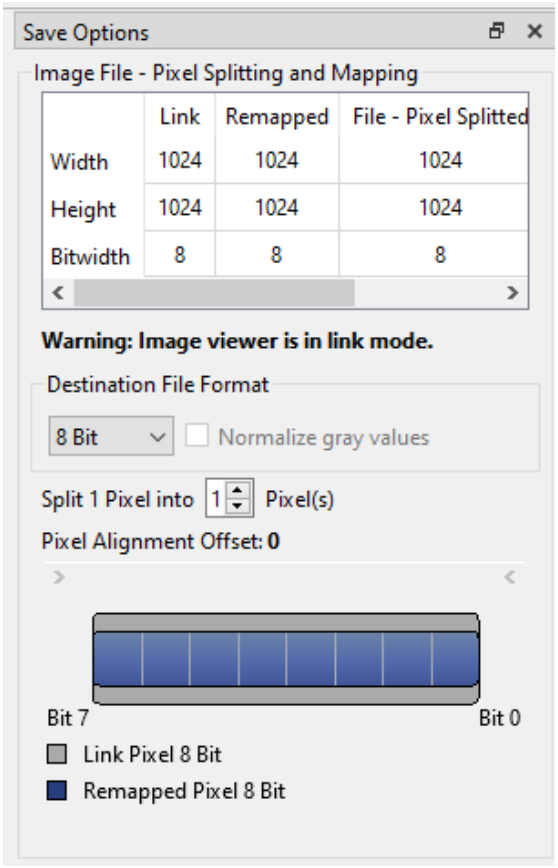



Figure 3.98. Save Options Dialog

In *Destination File Format*, you can define the format in which you want to save the image:

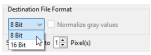


Figure 3.99. File Format Options for Saving

If you want to save a 1-bit simulation result in an 8-bit file format, check the box *Normalize gray values*. **Normalizing** in this context means mapping 0 to 0 (black) and 1 to 255 (white).

### 3.10.12.1. Pixel Alignment

If the bit width of the image (= the bit width of the link) is the same as the bit width of the file format you have chosen for saving, you can save the image 1:1.

If the bit width of the image is smaller than the bit width of the file format, you can change the pixel alignment offset using the slider. This way, you can save the bits of the simulation result on a certain location within the (larger) pixel of the file format. For example, save the bits of a 12-bit link on the lowest 12 bits or the highest 12 bits of a 16 bit TIFF image.

If the bit width of the image is larger than the bit width of the file format, you can use the slider for choosing which bits out of an image pixel you want to save to file and which bits can be discarded.

To get a preview on the (stripped) image as it will be saved to file: In the main window of the *Simulation Probe Viewer*, set the radio button on top of the settings panel to *File View*.

### 3.10.12.2. Pixel Splitting

If you want to use the pixel splitting option of VisualApplets, you can define to how many pixels you want to split the color information of one pixel (see also Section 3.10.6.5, 'Pixel Spitting and Merging') .

If you want to save, for example, a 16-bit simulation result in a 8-bit BMP file, set *Split 1 Pixel into* to 2 pixels. Now, the 16-bit pixel of the image is split into two 8-bit pixels.

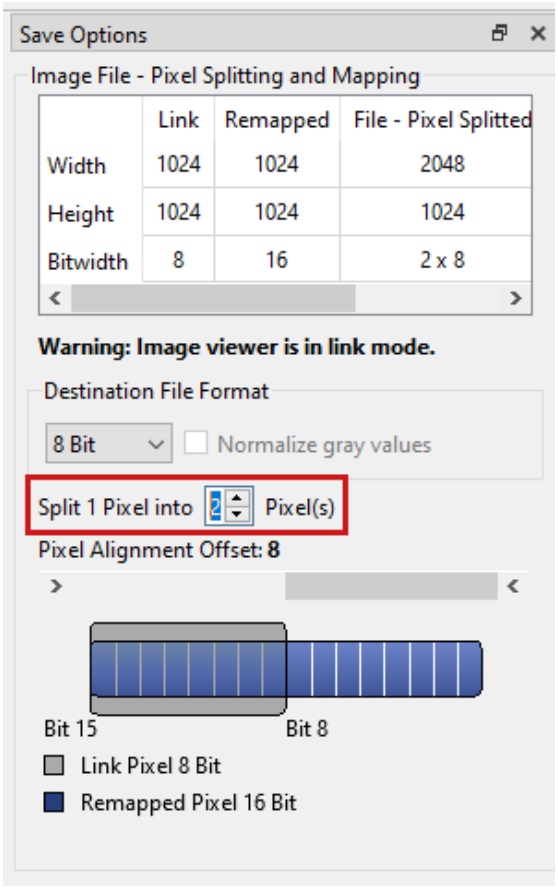



Figure 3.100. Setting the Splitting Factor in the Save Options Dialog

By splitting, the row length is doubled, tripled, etc. (depending on the splitting factor). In our example, the row length is doubled. Regular image visualization/processing programs can display these split images. But since they assume a bit width of 8 bit, the image display is expanded horizontally.

**Reconversion of Split Image**

You can load a split image in a simulation source and re-convert it to its original bit width by using the pixel merge option (see Section 3.10.6.5, 'Pixel Spitting and Merging').

### 3.10.12.3. Saving

**Important**

When selecting **File** -> **Save** from the main menu, all settings you have defined in the *Save Options* dialog are applied.

For saving your simulation results to file, proceed as follows:

1. Select from the main menu **File** -> **Save**.
2. Define saving location and file name.
3. Click **Save**.



**Saving an Image Sequence**

You can also save a whole image sequence. To save an image sequence, select all images you want to save in the *Simulation Viewer*. To do so, hold **CTRL** pressed and click on the images. Then, select from the main menu **File** -> **Save**. The saved images are automatically numbered.

### 3.10.13. Frequently Asked Questions

**Q:** I converted a color image to a gray-scale image and tried loading it to a simulation source. However, VisualApplets denies loading the image onto a one-channel link. What is the problem?

**A:** The image still uses 3 channels (with 0% saturation) for color information. Just convert your image to a one-channel image.

**Q:** My simulation probes do not include any results after simulation. What is the problem?

**A:** You might not have simulated enough simulation cycles. Increase the number of cycles. Moreover, your algorithm might not generate any output images. Check the respective operator restrictions in the operator references. Also, simulation probes connected to signal links never show output data.

**Q:** My simulation probes are cleared automatically. Why?

**A:** Each time the design is modified, the simulation probes might get cleared. This does not occur for minor changes of your design like moving modules or modifying internal parameters of operators that do not affect current ports. Probes are invalidated for changes that alter the topology of the design or parameter changes that are propagated through links to connected operators.

## 3.11. FPGA Resource Estimation

FPGAs contain a fix number of resources. These resources are components like BlockRAM, FlipFlops, LUTs, etc. An application design must be mapped into these resource components.

FPGA resources:

- BlockRAM is an FPGA-integrated memory used for saving FPGA internal data.
- FlipFlops are electronic circuits used for storing logical state information.
- LUTs (look-up-tables) are used for implementation of logic functions (AND, OR, XOR, etc.).
- RAM LUTs are LUTs-related FPGA internal memories.
- Embedded arithmetic logic units (ALUs or "embedded multipliers"") are hardware multipliers integrated into the FPGA.

How many resources you can use for implementing a design depends on the type of the FPGA.



**Resources of Supported Hardware Platforms**

In Appendix A, '*Device Resources*', you find information on all supported hardware platforms.

Every module of a VisualApplets design (except branch, trash, etc.) uses part of the available resources exclusively. Therefore, it is of utmost importance that you are informed about the current state of all resources in use while you are creating your design.



**No extra resources**

You can only use the resources available.

Information on resource usage of individual operators you find in the operator reference (see, for example, *MULT* or *DIV* in the Part III, 'Operator Reference', Arithmetics library).

### 3.11.1. Resource Usage Estimation on Design Level

The current resource usage of a design is in part calculated exactly and in part estimated by VisualApplets. Calculation and estimation are executed in Design Rule Check 2 (DRC2) (see Section 3.9, 'Design Rules Check').

Exact calculation of resource usage during Design Rule Check 2:

- BlockRAM
- Arithmetic logic units (hardware multipliers)

Estimation of resource usage during Design Rule Check 2:

- LUT
- RAM-LUT
- FlipFlops

Information on the exact resource usage of LUTs, RAM-LUTs and FlipFlops is only available after a successful build of an applet.

After Design Rule Check 2, the absolute result of the FPGA resource usage (followed by the percentage usage) for the complete design is displayed in the information panel, tab *Project Info*, section Project/ Resources (see Section 3.3.1.1, 'Project Info').

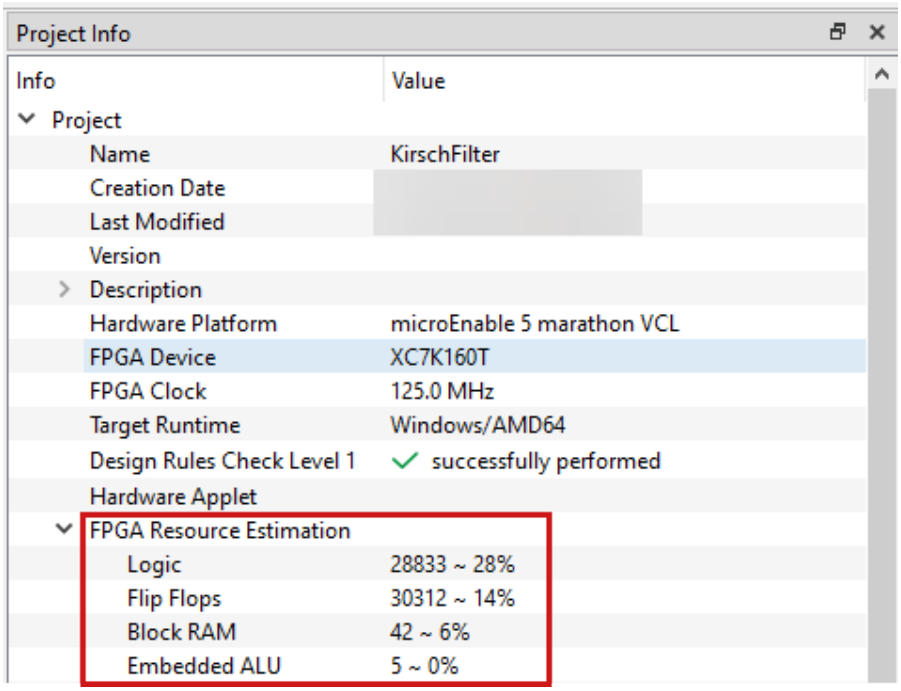



Figure 3.101. Project Info Window

The resource usage of a design depends on the parallelism (link parameter) used in the design, and on the number and complexity of the modules deployed.

**Keep Parallelism Low**

In order to save resources, you should always try to keep the parallelism of a design as low as possible when defining the settings for the required band width (see description of operator PARALLELdn in Part III, 'Operator Reference').

**Estimated Values and Successful Build**

If resource estimation values are slightly below 100%, build might fail. On the other hand, even if resource estimation values are slightly over 100%, build might be successful. This is due to the fact that the values are only estimated and the actual resource usage might be slightly higher or lower.

### 3.11.2. Resource Usage Estimation on Module Level

Detailed information on the resource usage of the entire design as well as on the resource usage of each individual module you can obtain by selecting from the main menu **Analysis** -> **View FPGA Resources** or by clicking the **View FPGA Resources** button .

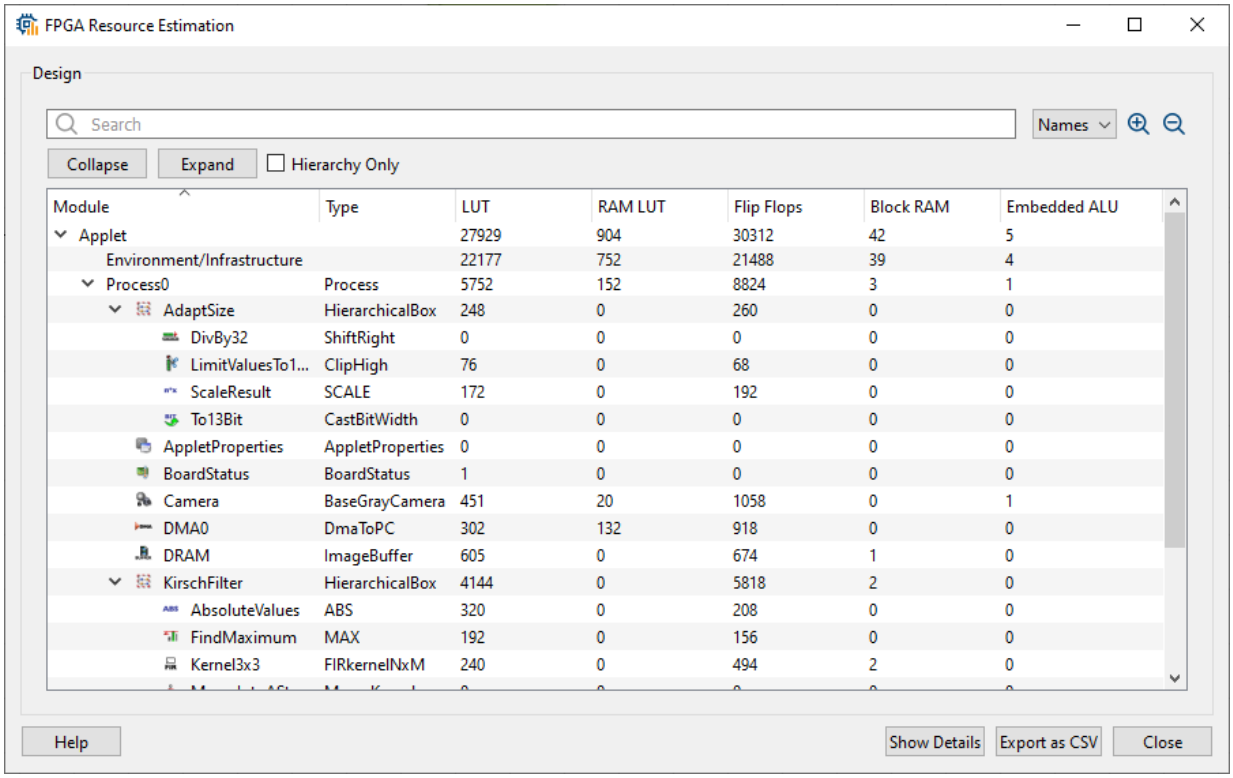



Figure 3.102. Detailed Information on FPGA Resource Estimation

For further comparison, you can export the estimated values as a coma-separated values (CSV) file by clicking the **Export as CSV** button.

You can access information on the estimated FPGA resource usage of an individual module via the context menu (right-click), selecting **FPGA Resources**.

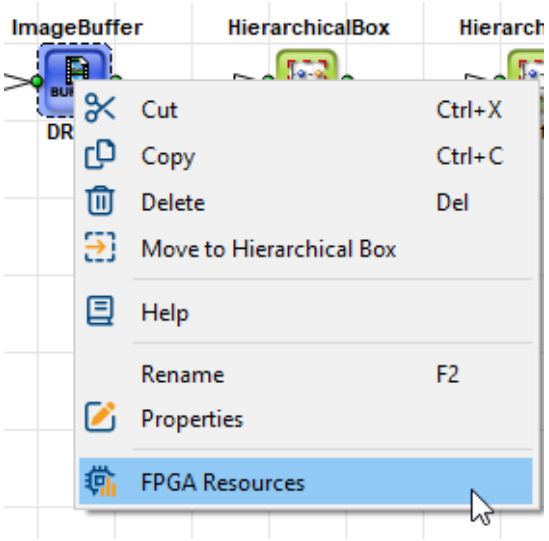



Figure 3.103. Context Menu *FPGA Resources*

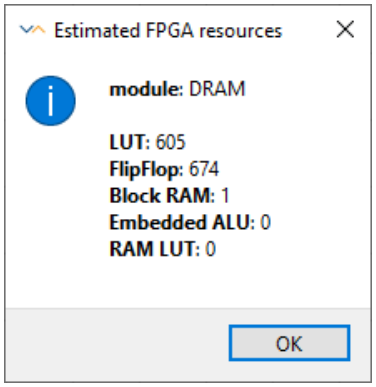



Figure 3.104. FPGA Resource Usage of Individual Module

## 3.12. Build

The applet build step is the final step in the design process of your applets. During the build step, the VisualApplets implementation is translated into the hardware applet (HAP). It is also called synthesis or compilation. The output of the build step is the hardware applet file *.HAP which can be loaded onto the frame grabber hardware.

VisualApplets will call the external tools of the FPGA manufacturer XILINX for HAP generation. Make sure these tools are installed in the right version and VisualApplets is configured accordingly.



**Xilinx Installation**

For a detailed description of the required Xilinx tools, and to learn all about their correct installation, refer to the VisualApplets Installation Guide [https://docs.baslerweb.com/visualapplets/installing-visualapplets].



**Build Settings for microEnable 5**

When designing for microEnable 5 or LightBridge, make sure you configured VisualApplets accordingly, see section Section 4.9, 'Build Settings'.

The *Project Info* dock window shows the version of the detected XILINX software (Figure 3.105, 'Detected Xilinx tools').

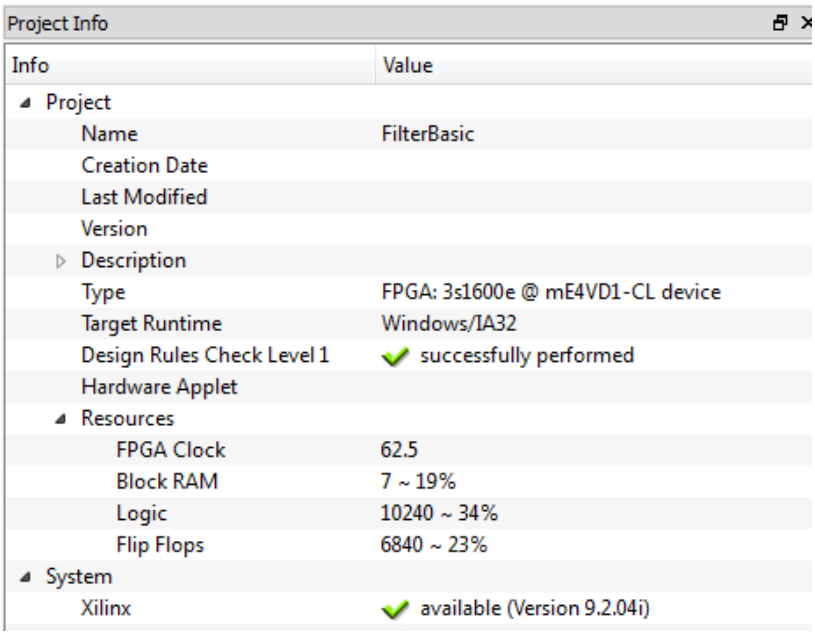



Figure 3.105. Detected Xilinx tools

Open the *Build Hardware Applets* dilaog by clicking **Build** -> **Build Hardware Applet**, by using the Build Harware Applet button  from the tool bar or by using the shortcut **F7**.



**Default Directory for Resulting Hardware Applet**

VisualApplets will not ask the user where to store the resulting hardware applet (*.hap file). The file is automatically generated and saved in the directory that is specified under Section 4.7, 'System Settings' */ Global Build Settings / Generation of Hardware Applets / Path for storage of hardware applets (*.hap)*. By default, this is the installation directory of VisualApplets.

If VisualApplets is installed on your system in a folder that requires administrator rights (e.g., C:\Programs), you have to start VisualApplets with administrator rights in order to use the default path, or to adapt the path setting under under Section 4.7, 'System Settings' / *Global Build Settings / Generation of Hardware Applets / Path for storage of hardware applets (*.hap)*.

The file is copied into the Framegrabber SDK installation directory, if a Framegrabber SDK instance is installed, matched with the design target runtime and recognized by VisualApplets. The output filename is equal to the filename of the *.va file, i.e., `myDesignName.hap`. The project name has no influence on the file name of the applet (*.hap file).

You can change the build settings. Read Section 4.9, 'Build Settings' for more information.

### 3.12.1. Selecting the Build Configuration

In VisualApplets, multiple build settings can be created in the in the *Build Settings* dialog which are stored in the application.

In the *Build Hardware Applets* dialog under *Xilinx configuration*, you can select one of the build settings you defined in the in the *Build Settings* dialog. The Build configuration you select here will be used for the build you are going to start. Default value is the configuration you specified as *Active configuration* in the *Build Settings* dialog.

1. Under *Xilinx configuration*, select the build configuration you want to use for the build.

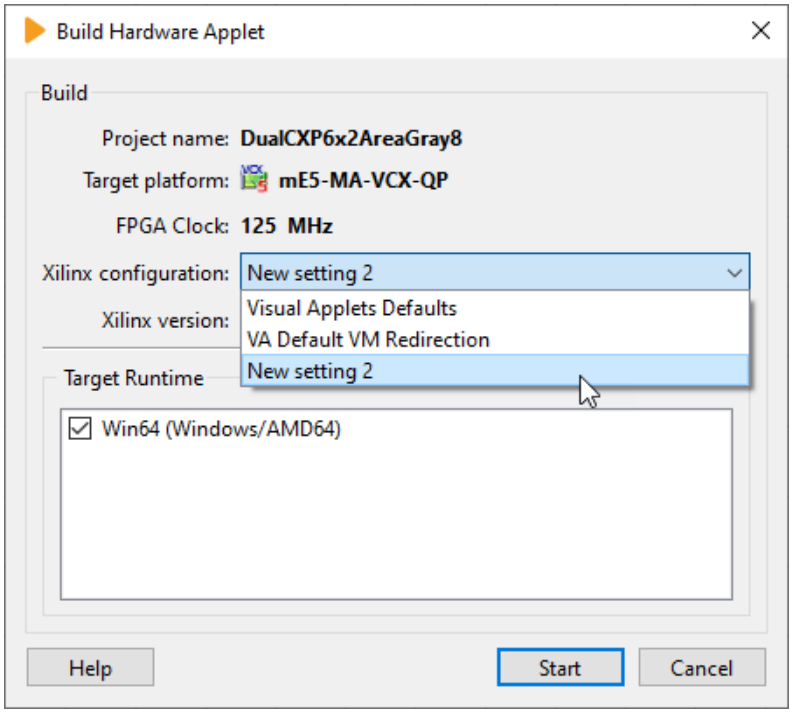



Figure 3.106. Selecting the Build Configuration for Applet Build

### 3.12.2. Target Runtime Selection

In the same *Build Hardware Applets* dialog, you can specify the required target runtimes. By default, the target runtime of the design file is selected. (See Section 4.8.1, 'Target Runtime') However, sometimes the applet will be used on multiple target runtimes. In this case, multiple target runtimes can be selected as can be seen in the next figure. VisualAppelts will generate a HAP file for all selected target runtimes.

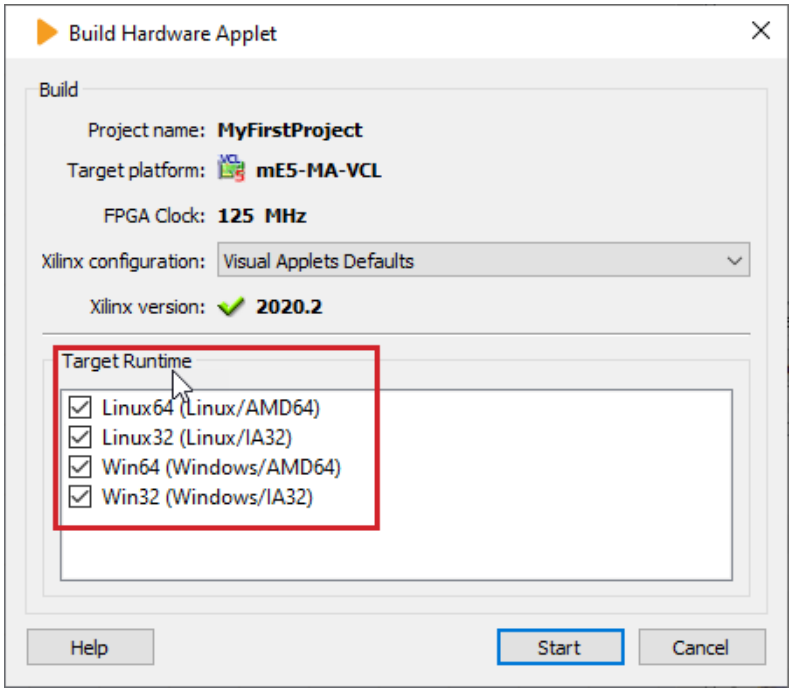



Figure 3.107. Target Runtime Selection during Applet Build

For the target runtime you defined the design file, the output file is located in the specified directory as described above. The file name of this output file is the same as the name of the *.va design file, only with the *.hap extention. The file is copied into the Framegrabber SDK installation, if the Framegrabber SDK is installed.

All additional output files: VisualApplets will add target runtime information to the file name of all additional output files, like "_Linux_AMD64", "_Linux_IA32", etc. These additional HAP files will **NOT** be copied into the Framegrabber SDK installation directory.



**Repacking the *.hap File for Other Target Runtimes at a Later Stage**

You can easily create a new *.hap file for an additional target runtime at a later stage without starting the whole build process again. This may save you a lot of time as the repacking only takes some minuts. For details on how to repack a *.hap File for additional target runtimes, see section Section 3.12.7, 'Repacking the *.hap File for Other Operating Systems at a Later Stage'.

### 3.12.3. Build Settings for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta

The platforms imaFlex CXP-12 Quad and imaFlex CXP-12 Penta have one additional build setting: **Netlist synthesis engine**.

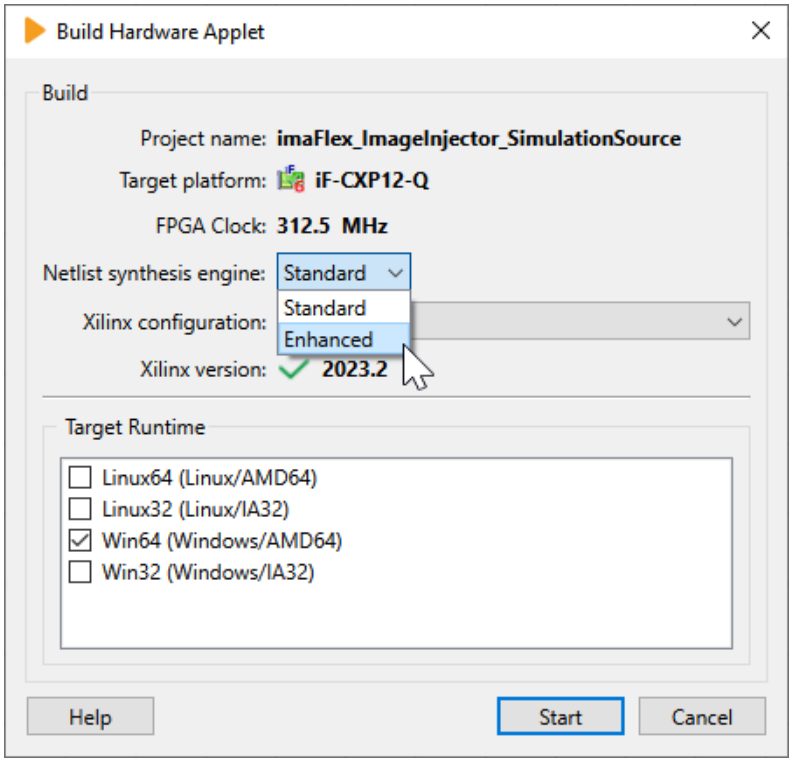



Figure 3.108. Build Setting for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta

This setting has an effect on the build velocity and device ressources.

**Standard**:

- Builds your design in approximately one hour.
- Does provide a ressource estimation after the build.
- Does **NOT** work, if you have operators of the *Blob* library in your design. In this case, the build will be aborted with an error message.

**Enhanced**:

- Builds your design in approximately three hours.
- The built design uses less ressources.
- Does **NOT** provide a ressource estimation after the build.
- Works for all operators.

Basler recommends to use always the **Enhanced** netlist synthesis engine.

### 3.12.4. Errors during Build

The build process will always execute design rules checks level 1 and level 2. If one of the fails, the build process is aborted. Check Section 3.9, 'Design Rules Check' for more information on the design rules checks.

Moreover, the build process can cause several further errors. The most likely errors is a resource overmap or timing error. If an applet contains to much logic elements, the XILINX tools cannot map all required elements to the physically available elements. In the case of a resource overmap, your design has to be optimized to fit into the FPGA.

In case of a timing error, the XILINX tools cannot route all signals within the allowed constraints. This mostly happens if almost all logic resources are used. The design has to be optimized to use less resources in this case. Designs which have a timing error can still be completed. **Keep in mind that a hardware applet which did not meet timing might not work correct. It might result in time-outs, failing data or other unexpected behaviors if used. Therefore, the use of a non-timing matched applet should only be used with care and is at the user's own risk.** If the required resources of your design are far less than 100% and you still get timing errors, contact the Basler Support [https://www.baslerweb.com/en/sales-support/support-contact/].

Other errors can be caused by an incorrect XILINX installation or disk volume access permissions.

Note that the build process might require some hours to be completed.

### 3.12.5. Applet Run

The usage and run of a hardware applet file (HAP) is described in the Framegrabber SDK documentation [https://docs.baslerweb.com/frame-grabbers/managing-applets-micro-diagnostics]. The SDK generator of VisualApplets can help with the first steps of running an applet. See Section 3.13, 'Framegrabber SDK' for more information.

### 3.12.6. microDisplay

microDisplay is part of the Framegrabber SDK. It is a tool for first tests of the generated applets. If the Framegrabber SDK is installed, microDisplay can directly be opened from VisualApplets by clicking on **Build** -> **microDisplay** ( **F5** ). Alternatively, you can also use the icon microDisplay from the Build icon bar.

### 3.12.7. Repacking the *.hap File for Other Operating Systems at a Later Stage

You can create additional *.hap files out of an already existing *.hap file. This is especially helpful if you want to use an applet on additional operating systems you did not specify before the first build.

You do not need to start the whole build process again. This saves you a lot of time. The repacking only takes some minutes. Only the operating system specific parts of the original *.hap file are replaced by VisualApplets during the repacking process.



**Pre-Conditions**

To repack an existing *.hap file for a new operating system, you must use the same VisualApplets version that was used to build the original applet (VA 2.2 or higher).

Otherwise, you will get an according error message.

To create additional *.hap files for additional operating systems:

1. From the menu, select **Build** -> **Repacking Hardware Applet Files...**.

   The *Repacking Hardware Applet Files* window opens:

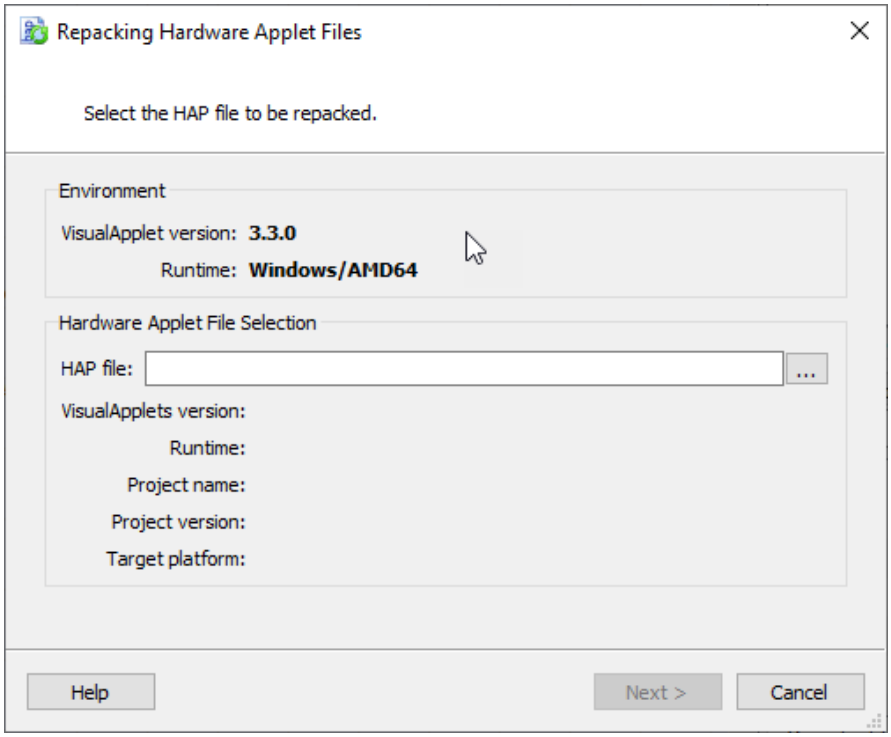



Figure 3.109. Repacking Hardware Applet Files Window

2. Select the original *.hap file using the browse button.

   VisualApplet immeadiately checks if the original *.hap file was created with the same version of VisualApplets (2.2 or higher) you are using now (which is a precondition for the repacking process, see above).

   If the VisualApplets version doesn't match, an according error message is displayed.

3. If everything is fine, click the *Next* button.

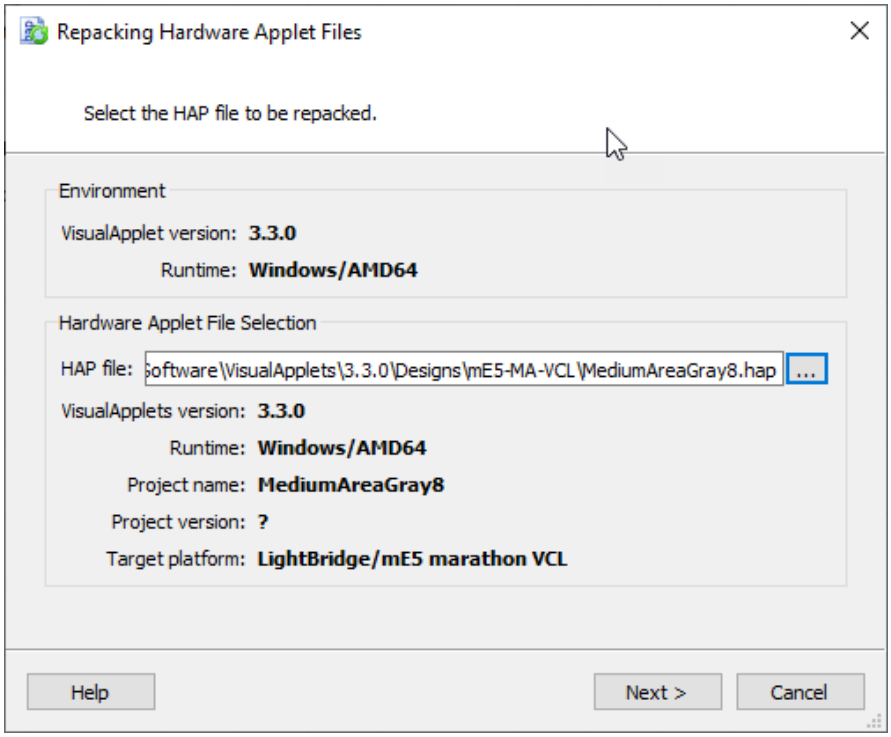



Figure 3.110. Fullfilled Repacking Preconditions

4. Under *Runtime*, select the target runtime (i.e., target operating system) you want to create the new *.hap file for.

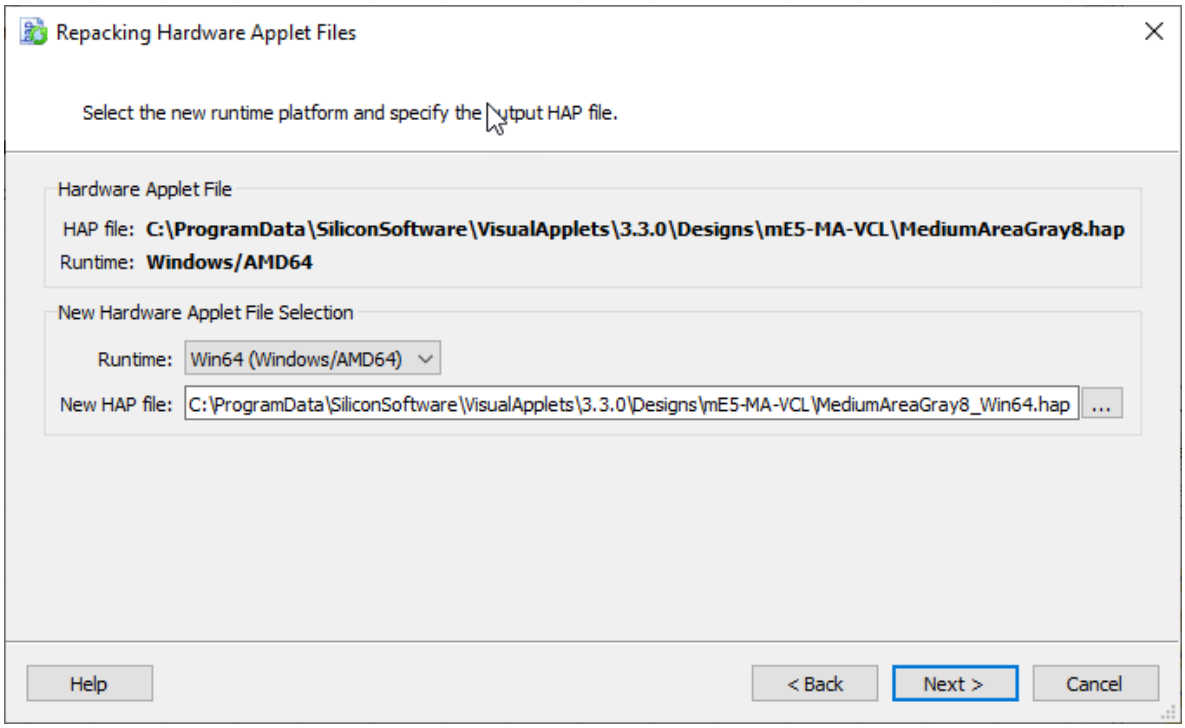



Figure 3.111. Selecting Target Operating System

VisualApplets offers a file name for the new *.hap file.

5. Adapt the output location and the file name for the new *.hap file to your needs.
6. Click *Next*.

   VisualApplets displays all settings you have specified for the repacking:

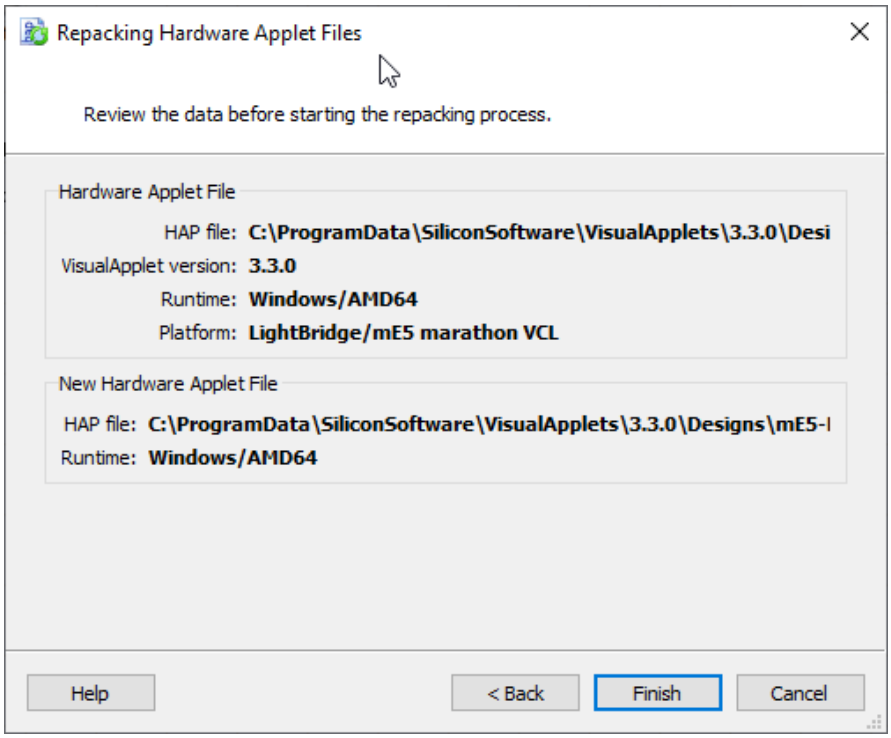



Figure 3.112. Display of Specified Repacking Settings

If you need to adapt your settings, you can use the *Back* button.

7. Click *Finish* to start the repacking process.

   After successful repacking, you get an according message:

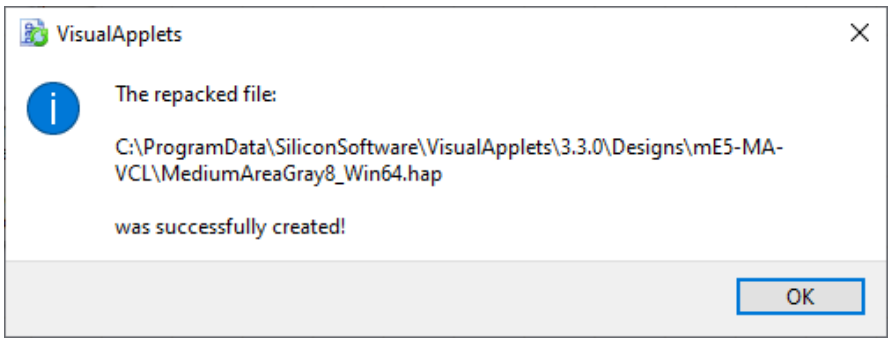



Figure 3.113. Message after Successful Repacking

# 3.13. Framegrabber SDK

The Basler Framegrabber SDK is used to access the hardware devices in the runtime environment. All VisualApplets implementations can be accessed using the API functions of the Framegrabber SDK. VisualApplets can generate an individual SDK sample project for each VisualApplets implementation.

## 3.13.1. Generating an SDK Example

You can generate a Framegrabber SDK example code for each VisulApplets project. To generate an SDK example, select **Build** -> **Generate SDK example** from the menu or select the  icon in the *Build* toolbar to generate the example. The *Browse to Folder* dialog opens up.

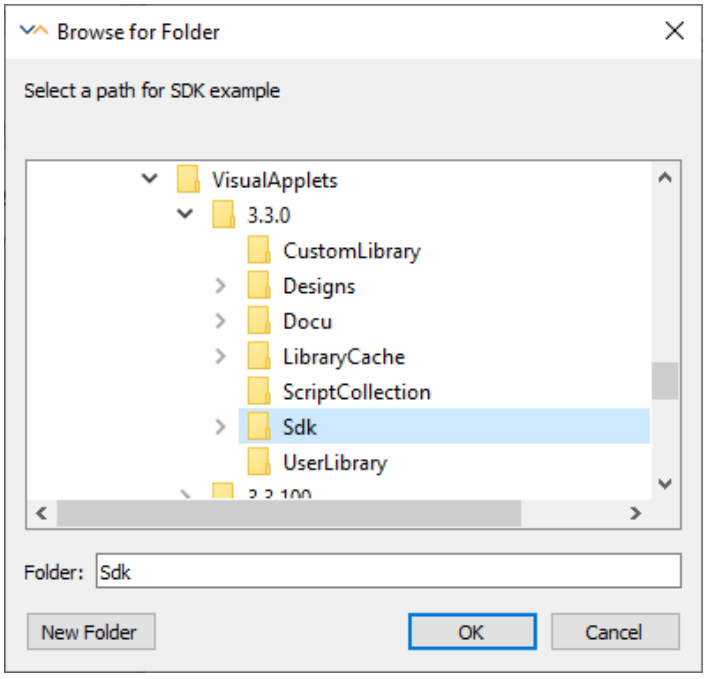
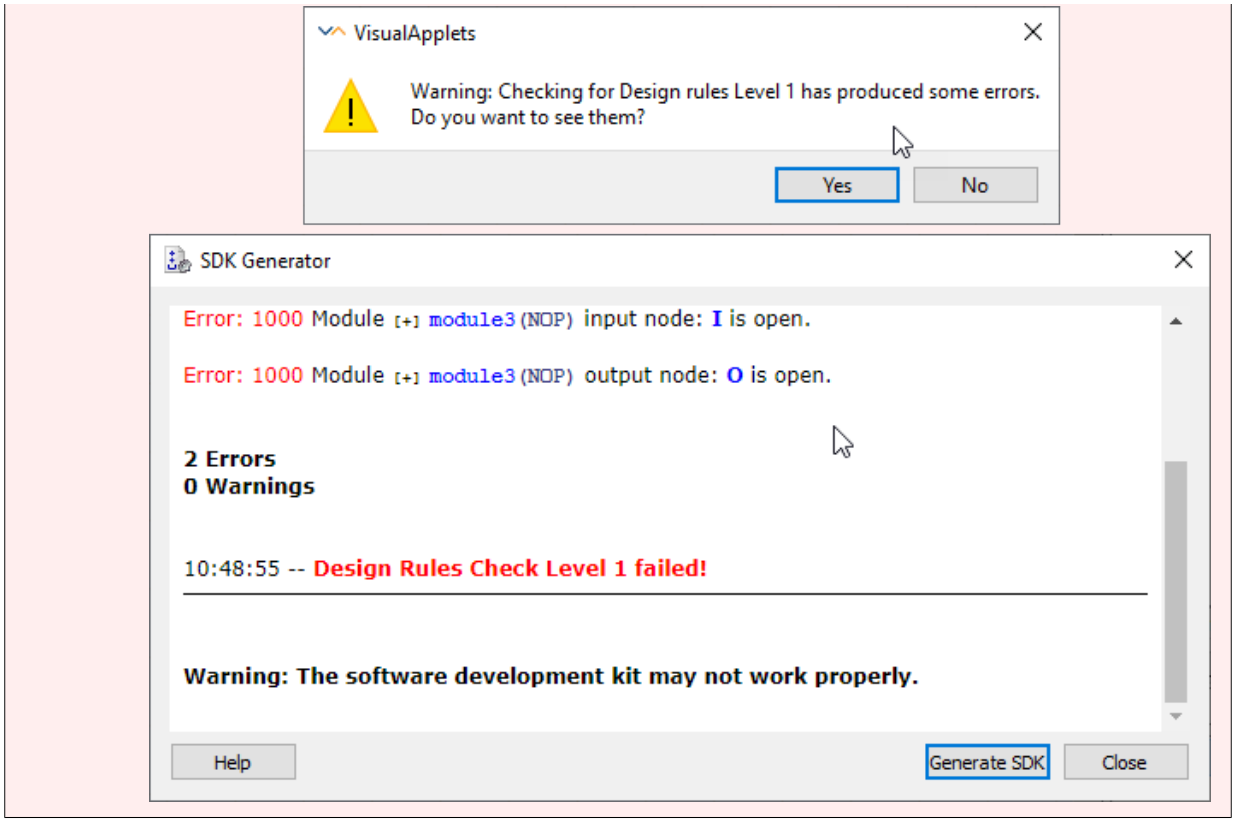



Figure 3.114. Selecting the Storing Location for the SDK Example

Select the location where you want your SDK example to be stored and click **OK**. Note that no subdirectory is generated, therefore it is recommended to create a subdirectory. The SDK example is generated and stored in the selected directory.

**VisualApplets Warns in Case of Erroneous Designs**

Before each SDK example generation, VisualApplets runs a DRC check and prompts a warning, if the DRC check fails:




### 3.13.2. Using the Generated SDK Example

The SDK example includes the source files as well as a Microsoft VisualStudio 6 project file which you can open also in newer versions of VisualStudio. You can compile and execute the example immediately. All available parameters are included and parametrized examplarily.

For a detailed explanation on the usage of a VisualApplets in the Framegrabber API, refer to the Framegrabber API documentation [https://docs.baslerweb.com/frame-grabbers/framegrabber-api]. Use the function `Fg_getParameterIDByName()` to obtain the unique parameter ID as shown in the example. A header file is generated for all enumerations in VisualApplets operators. A namespace with the name of the operator is created for each operator type used in the applet. Multiple enumerations are embedded in this namespace.

# 4. Extended Functionality

In this chapter, you are introduced to the possibilities and functions VisualApplets offers to the advanced user. This comprises, amongst other options, a clear structuring of larger designs, creating your own libraries, creating multiple-process designs, defining settings for design and build, and working with different program versions.

## 4.1. Hierarchical Boxes

The *HierarchicalBox* operator is a special operator in VisualApplets that allows to structure designs. As the name implies, the operator has the function of a box. You can mark a part of your design and make it the content of a hierarchical box.

The hierarchical box is visible in the design as an operator instance (of operator *HierarchicalBox*). The content of the hierarchical box you can see and edit in an individual design window.

A hierarchical box contains a combination of operators and links and can connect them to an arbitrary number of input and output links. The use of *HierarchicalBox* modules allows you to

- structure your design very clearly, and to
- administrate groups of modules.

In the design panel, the *HierarchicalBox* module is displayed as an M-type operator (oval shape); nevertheless, its behavior depends on the operators inside the hierarchical box.

The following figure shows an example of a *HierarchicalBox*. As you can see, the operators included in the hierarchical box are displayed in a separate design window. Depending on the window settings, both diagrams can be viewed at the same time as shown in the example. See Section 3.3, 'Main Program Window' for more information on how to arrange diagram windows.

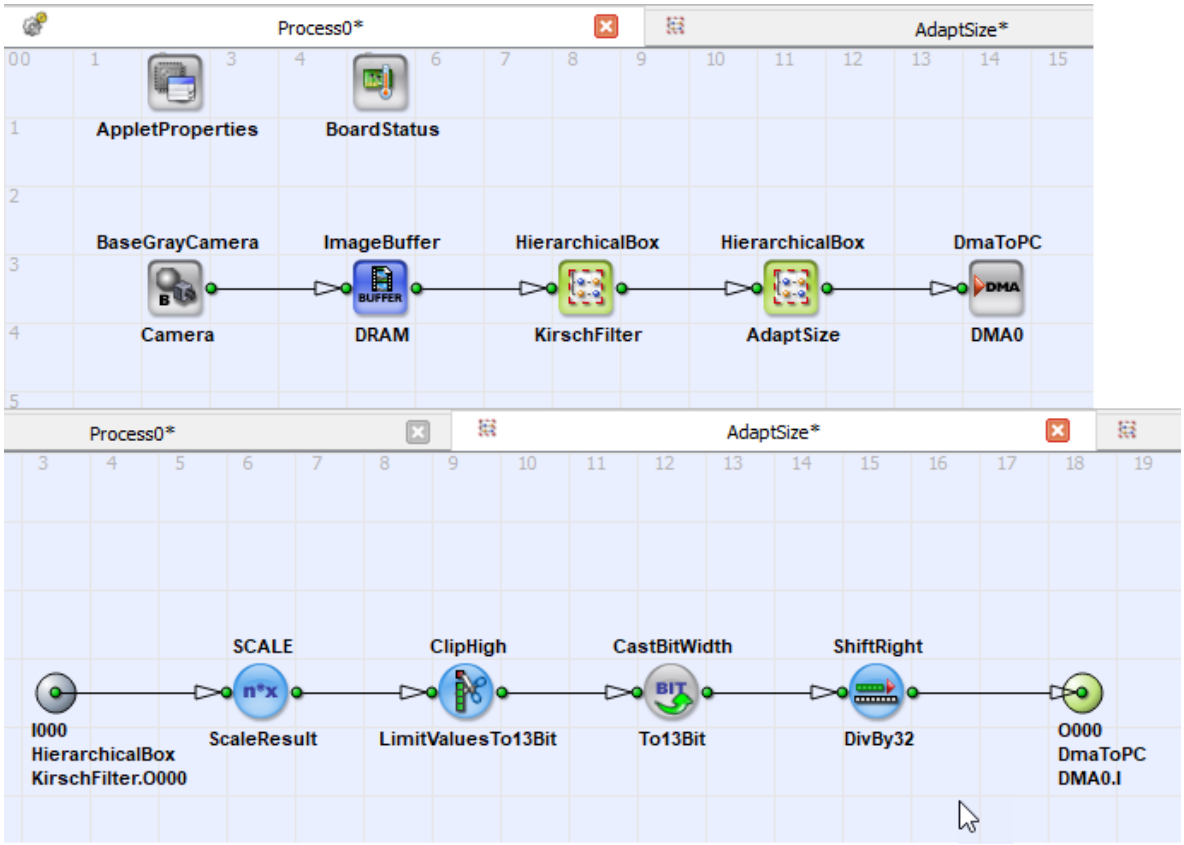



Figure 4.1. Example of a Hierarchical Box

### 4.1.1. Creating a Hierarchical Box

There are several ways to add a hierarchical box:

- Making a part of your design a hierarchical box:

  1. Select the modules of your design you want to assemble as hierarchical box. (Press the STRG key on your keypad and click on the modules you want to assemble.)

  2. Keep the STRG key pressed and right-click on an empty part of the design window to open the context menu.

  3. From the context menu, select **Move to Hierarchical Box**.

     A new hierarchical box is created that contains all selected modules and the according in- and out links.

  4. Double-click on a hierarchical box to open the diagram window that shows the hierarchical box implementation.

- Defining an empty hierarchical box:

  1. Select the *HierarchicalBox* operator from the *Base* library and add it to your design, or

     Right-click on an empty area in a diagram and select **Insert** -> **Hierarchical Box**, or

     select **Design** -> **Hierarchical Box** from the main menu, or

     Select the icon  in the *Edit* toolbar.

2. In the window that occurs, define the number of input and output links of your new *HierarchicalBox*.
3. Click **OK**.

   The *HierarchicalBox* module is inserted into your diagram; it has the number of input and output ports you just specified.
4. Double-click on the hierarchical box module to open its design window.
5. Add the operators you want to have in your hierarchical box.
6. Connect the design in the box to the input and output ports of the box via links.

### 4.1.2. Navigating between Design Windows

To go back to the design window of the process that contains the hierarchical-box module:

1. Click on the window tab in the design window:




Figure 4.2. Window tabs of the design window

To step back in the hierarchy:

1. Select **View** -> **Window Up** ( **Backspace** ) or click the *Window Up* button  in the *View* toolbar.

### 4.1.3. Editing Module Properties

To open the module properties:

1. Right-click on a hierarchical box and from the context menu, select "Properties". Here, you can see and edit the parameters of the hierarchical box that are available for parametrization.

For information how to make parameters deeply nestled within a protected hierarchical box (or within a protected hierarchical box that is part of another protected hierarchical box) visible and available at this level, see the Operator Reference, library "Parameters".

### 4.1.4. Re-use of Hierarchical Boxes

You can use hierarchical boxes more than once by copy & paste. Each instance of a hierarchical box is independent from its siblings,i.e., if you make changes to one of the copied hierarchical boxes, this has no impact on the others.

For information on how to use hierarchical boxes as library elements and protected library elements, see section Section 4.2.2, 'Creating a User Library Element'.

### 4.1.5. Inserting Additional Ports to Hierarchical Boxes

You can add further input and output ports to your hierarchical box anytime. To add a port:

1. Navigate to the design window that shows the hierarchical box module.
2. With the mouse pointer, go to an input port (if you want to insert an input port), or to an output port (if you want to insert an output port) so that the port is highlighted.

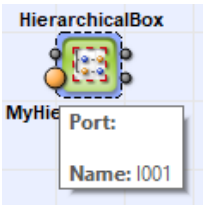



Figure 4.3. Highlighting a Port

3. Right-click on the port.
4. From the context menu, select **Insert Port Above** or **Insert Port Below**.

The new port is inserted. It is given a default name. To give an individual name to the port, use context menu item **Rename Port** (see below).

### 4.1.6. Deleting a Port of a Hierarchical Box

You can delete input or output ports of your hierarchical box anytime. To delete a port:

1. Navigate to the design window that shows the hierarchical box module.
2. With the mouse pointer, go to the port you want to delete so that it is highlighted.
3. Right-click on the port.
4. From the context menu, select **Remove Port**.

The port is deleted.

### 4.1.7. Re-naming the Ports of Hierarchical Boxes

To enhance the readability of your design, you can give names to the input ports and output ports of a hierarchical box:

1. Navigate to the design window that shows the hierarchical box module.
2. With the mouse pointer, go on the port you want to re-name so that it is highlighted.

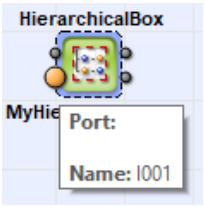



Figure 4.4. Highlighting a Port

3. Right-click on the port you want to re-name.
4. From the context menu, select **Rename Port**.
5. In dialog **Rename Port**, enter a name for the port.

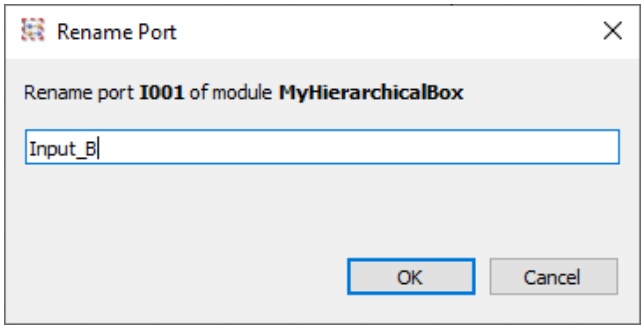



Figure 4.5. Entering a port name

6. Click **OK**.

The new name is visible, for example, in the design window of the hierarchical box module:

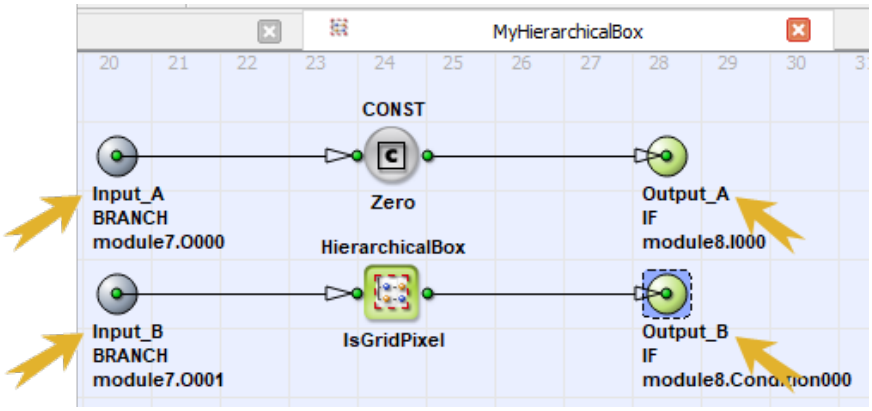



Figure 4.6. Renamed ports of a hierarchical box

## 4.1.8. Re-ordering the Ports of Hierarchical Boxes

In quite the same manner as renaming, you can re-order the ports of a hierarchical box module:

1. Navigate to the design window that shows the hierarchical box module.
2. With the mouse pointer, go on the port you want to change the position of so that it is highlighted.

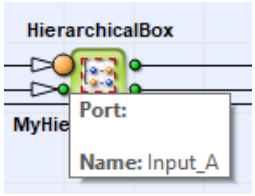



Figure 4.7. Highlighting a port

3. Right-click on the port you want to re-order.
4. From the context menu, select **Move Port Up** or **Move Port Down**.

The order of the ports is swapped. This is visible, for example, in the design window of the hierarchical box module:

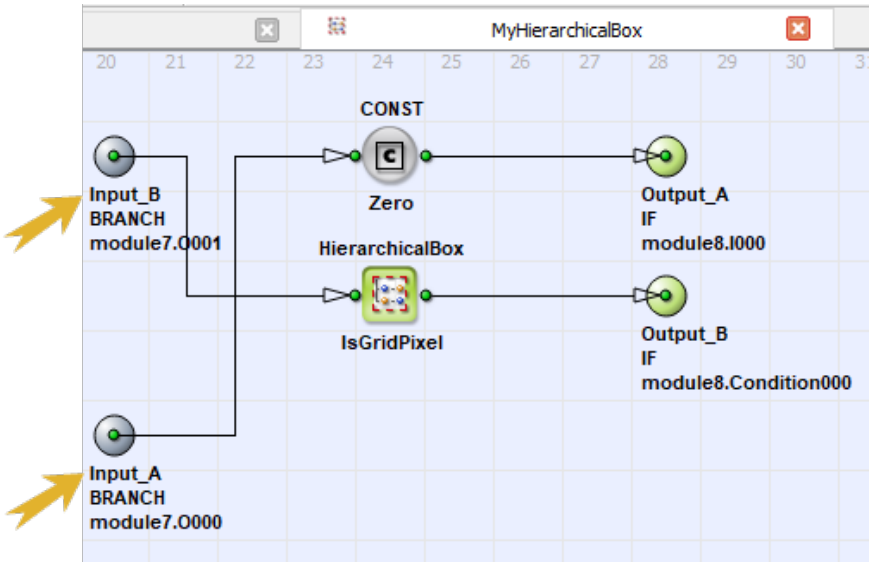



Figure 4.8. Reordered input ports of a hierarchical box

The re-arrangement of port order has no impact on the functionality of the hierarchical box. The re-arranged ports are linked to the same inner and outer design elements as before.

## 4.2. User Libraries

In user libraries, you can encapsulate a combination of modules which you need more than once and store this combination for later re-use.

You can define as many user libraries as you want. Each user library can contain as many user library elements as you want.

A **user library element** contains a specific combination of modules and links. After you create them, library elements are available in the library panel (tab *User Library*) and can be used like operators.

A user library element is pretty much the same as a hierarchical box (see Section 4.1, 'Hierarchical Boxes'). The difference is that user library elements are stored in a (user-defined) user library and can be inserted into different designs and projects, just as operators from the operator libraries.

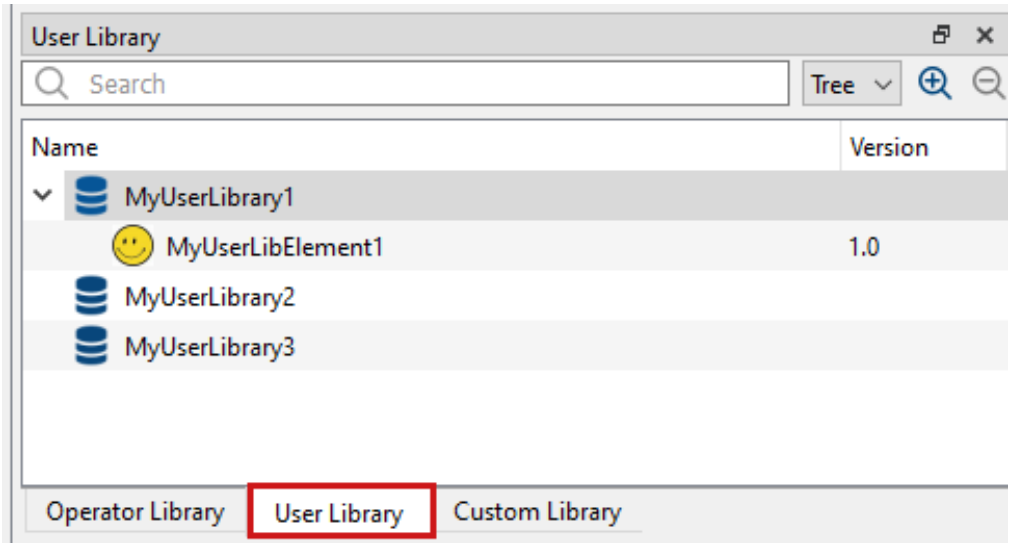



Figure 4.9. User Libraries with Elements in the Library Panel

The instances of a user library element are independent from each other, that is, if you make changes to one of the instances, this has no impact on the other instances of the same user library element.

User libraries are saved as `*.val` or `*.vl` files. These files are not part of a particular project. Thus, they can also be used in other projects.

In the *System Settings* of VisualApplets, you can specify where you want the `*.val` or `*.vl` files to be stored on your system (see Section 4.7, 'System Settings').

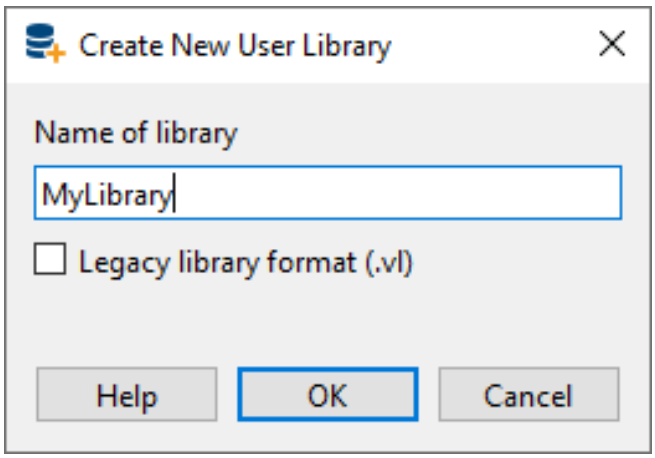
When you have inserted the instance of a user library element into a design, you do not need the `*.val` or `*.vl` file for loading the project or for building the applet (*.hap file).

### 4.2.1. Creating a New User Library

To create a new user library, proceed as follows:

1. In the library panel, click the *User Library* tab.
2. Right-click in the user library window.
3. From the context menu, select **Create New User Library**.

   (Alternatively, you can also use the main menu **Library** -> **Create New User Library**.)
4. The **Create new User Library** dialog opens up:



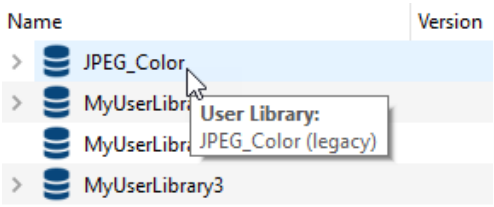

5. Give a name to your new user library.

6. By default, the new user library is saved in the `*.val` file format. If you want to open this library in VisualApplet version 3.3.2 or older, select **Legacy library format (.vl)**. Note that if your legacy user library in `*.vl` file format contains a protected user library element, this element isn't saved and can't be opened in the legacy file format.

7. Click **OK**

Now, an empty user library (functioning like a container) has been created which you can fill now with your user library elements.

When you move the mouse over the user library name, a tooltip shows whether the library is saved in the `*.vl` legacy file format:




## 4.2.2. Creating a User Library Element

There are two ways to create a new user library element. You can either

- save a hierarchical box as a user library element (for general information about hierarchical boxes, see Section 4.1, 'Hierarchical Boxes'), or
- create a user library element from scratch (see Section 4.2.2.2, 'Creating a New User Library Element from Scratch').

### 4.2.2.1. Saving a Hierarchical Box as a User Library Element

1. Right-click on a hierarchical box to open the context menu of the module.
2. Select **Save to User Library**.

   A *Save* dialog pops up:

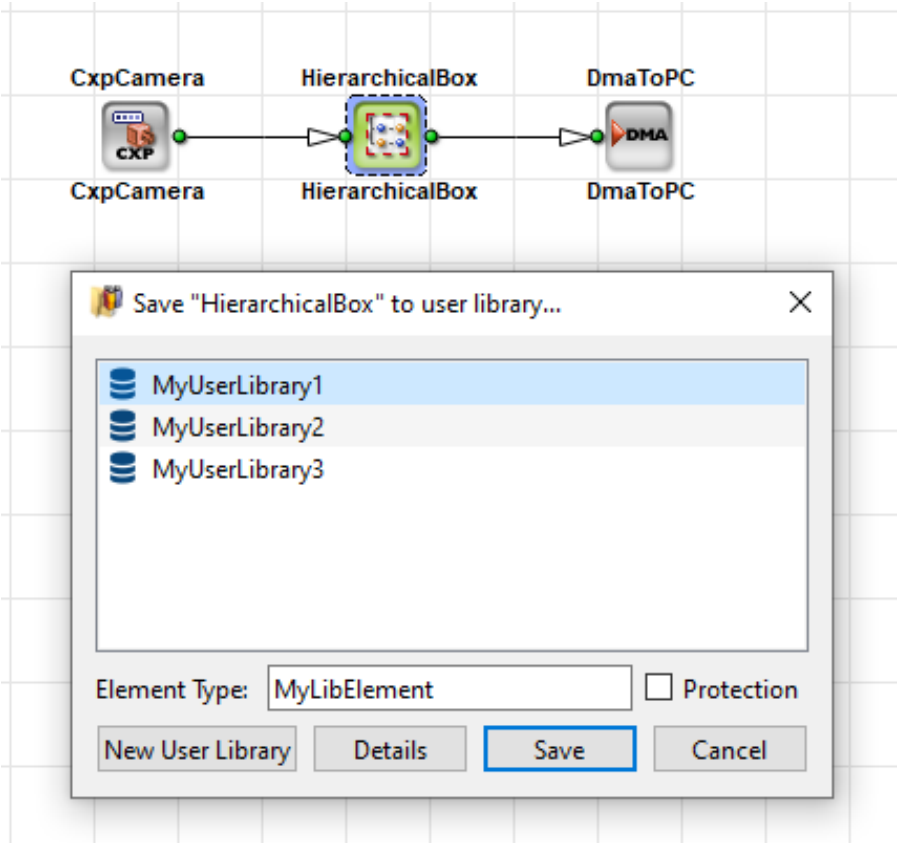



Figure 4.10. Saving a Hierarchical Box as a User Library Element

3. Select the user library you want to save the new element in (or, alternatively, click on **New User Library** and define a new library).

4. Enter a name (under *Element Type*) for your new user library element.

5. Optionally: If you want to add information on your element (version number, short description, individual GUI Icon, documentation in form of HTML help files):

   a. Click on **Details**. A new dialog opens:

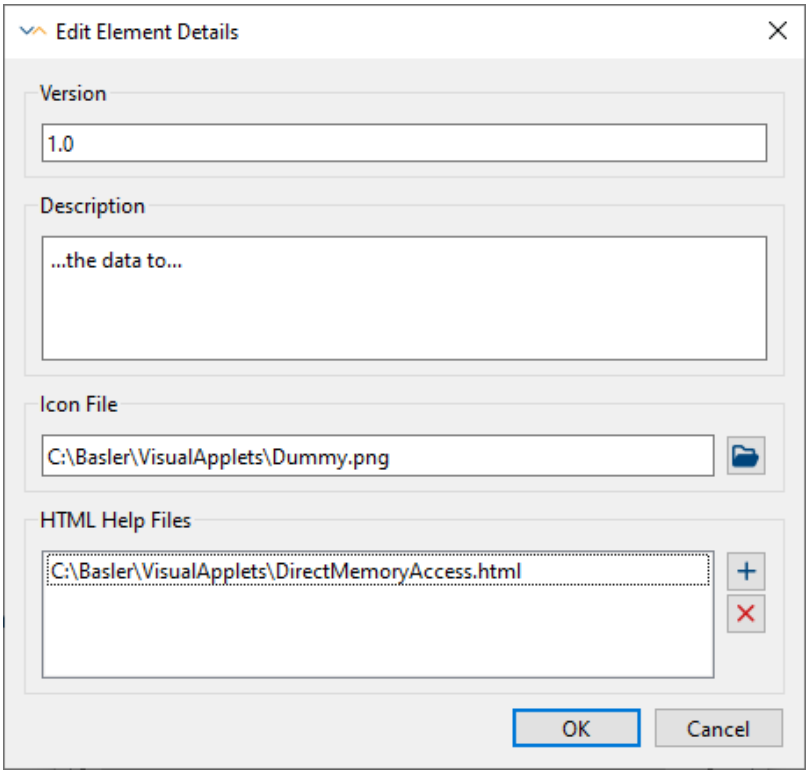



Figure 4.11. Adding documentation, version information, short description, and/or individual GUI Icon

b. Enter a version number for your library element if you want to.

c. Enter a short description for your library element if you want to.

d. To specify a specific icon to be displayed together with the library element on the VisualApplets GUI, in field **Icon File** specify the path to the PNG file or use the navigation button on the right hand side of the field.

e. To add documentation describing your user library element, click the plus button under **HTML Help Files** and navigate to the HTML file that holds the text of your documentation.

   You may specify multiple HTML and image files. If you add multiple HTML files, the first HTML file in the list will be the starting page of your documentation. The starting page is displayed as documentation of your user library element (in the information panel, tab **Help**). The additional files may contain images or further HTML files that are linked by the starting page.

   **Naming and Storage of Help Files**

   When you save your user library element later on, all the files you specify here will be copied to the VisualApplets installation directory, sub-directory **LibraryCache**. The starting page of your documentation (first HTML file in the list) will be renamed and will have the name of the user library element:**<elementname>.html**

   To unlink an HTML file from the library element, select the file and click on the delete button. The HTML file will be unlinked, but remains in your file system.

f. Confirm the details specification of your new user library element with **OK**.

6. If you want to protect your user library element, select the *Protection* option. This way, the user library element is made a “black box”. (For details on element protection, see Section 4.2.4, 'Protecting User Library Elements'.)

7. Click **Save**.

   The optional PNG and HTML files are copied to the VisualApplets installation directory, sub-directory **LibraryCache**.

   If you activated the *Protection* option, an additional dialog *Protect User Library Element* appeares.

   a. Make sure protection mode **Password** is activated.

   b. Enter your password and click **OK**.

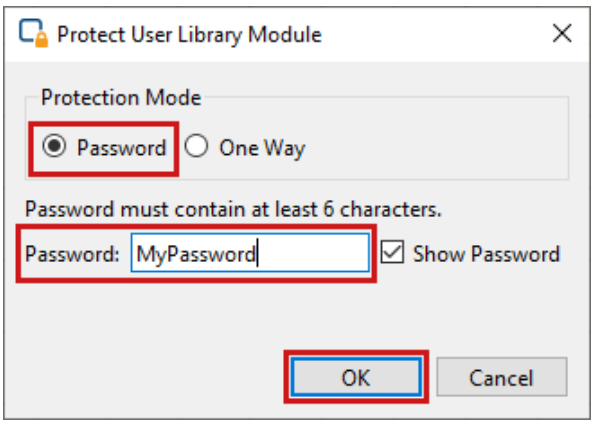



Figure 4.12. Providing a password for a library element

All User libraries and their contents are available in the library panel (see Section 3.3, 'Main Program Window'). Click on the *User Library* tab to display the User Library window.

The icon you selected is visible together with the library element.

Detailed information on an individual element is displayed in the tooltip:

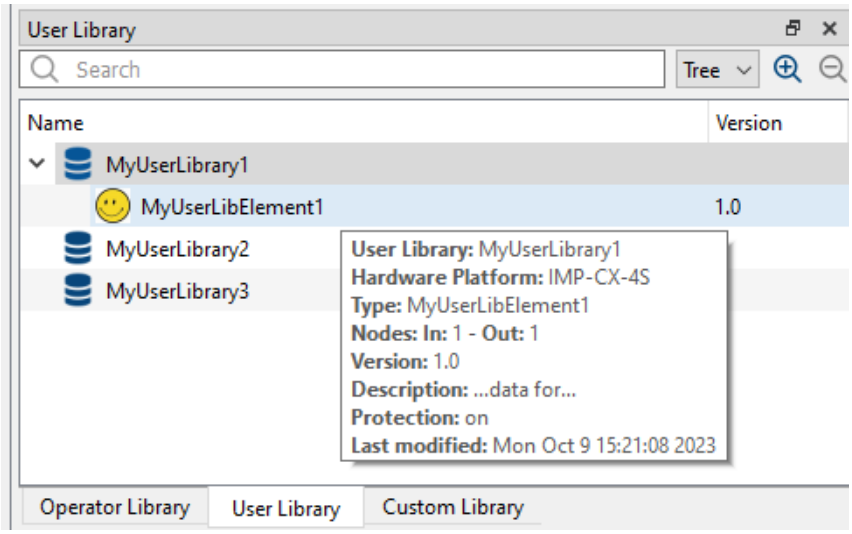



Figure 4.13. Tooltip Information on User Library Element

If you select the new user library element, the according help information is displayed in the information panel under tab **Help**.

The display within the design window has changed: Instead of "HierarchicalBox", the name of the user library element is displayed. The icon for hierarchical boxes has been replaced by the icon you selected for the library element. The name of the instance within the design (below the icon) has remained the same.

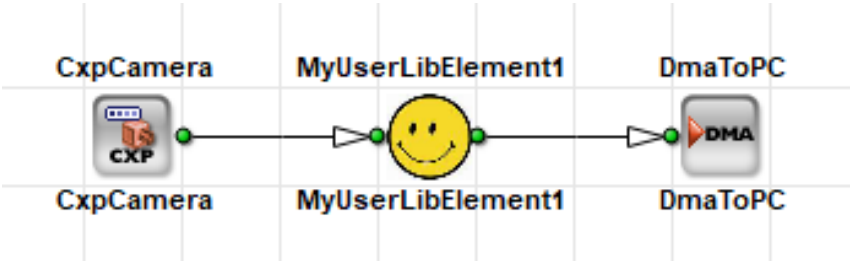



Figure 4.14. Display of Your Library Element in Design

You can change your user library element any time later on (see section Section 4.2.5, 'Editing User Library Elements'. However, these changes will not be promoted to the instances of the element. You have various options for updating the instances of a user library element if you want to (see section Section 4.2.6, 'Updating Instantiated User Library Elements'.

**Analysis of User Library Elements**

Design Rules Check Level 1 is possible.

Resource analysis, build, and simulation are not possible.

### 4.2.2.2. Creating a New User Library Element from Scratch

Alternatively, you can create new user library elements from scratch. For defining and/or changing user library elements, you use the **User Library Editor**.

To define a new user library element from scratch:

1. From the main menu, select **Library** -> **Edit User Library** -> **New User Library Element**.

   Now, the design editor closes. Together with the design editor, also the design you have been working on is closed.

   The user library editor opens. The user library editor behaves like the normal design editor.

2. Define a name for your new user library element.

3. Select the target hardware platform.

   (However, the platform is only relevant if you include operators in your element that are dependent on hardware issues, such as camera operators.)

4. Click **OK**.

5. As soon as you click **OK**, a dialog box opens where you have to specify the number of input and output ports of your element. Make sure you specify the right number of input and output links since it is not possible to change these numbers later on.

   a. Specify the number of input ports you need.

   b. Specify the number of output ports you need.

**Define necessary number of links**

Make sure you specify the right number of input and output links since it is not possible to change these numbers later on.

6. Confirm the port definition with **OK**.

   The user library editor opens with an empty element design containing only the imput and output ports you just specified.

7. Edit the design of your user library element as you want.

8. When you are finished, save your element by selecting from the main menu **File** -> **Save As**.

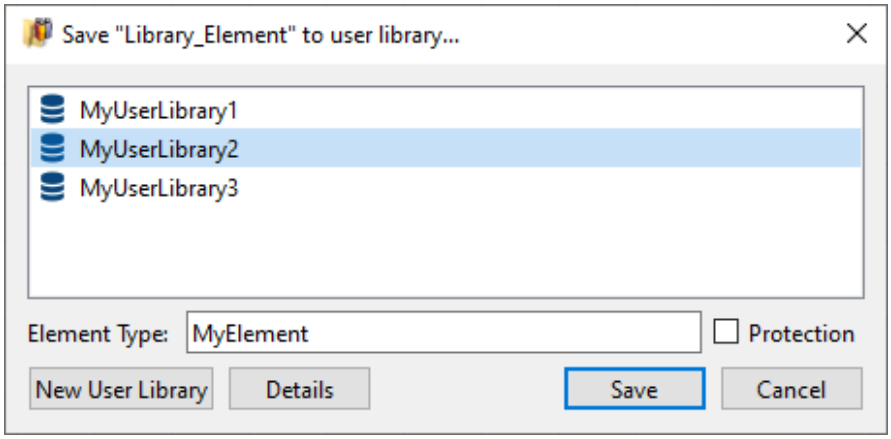



Figure 4.15. Saving New User Library Element

9. Select the user library you want to save the new element in (or, alternatively, click on **New User Library** and define a new library).

10. Optionally: If you want to add information on your element (version number, short description, individual GUI Icon, documentation in form of HTML help files):

    a. Click on **Details**. A new dialog opens:

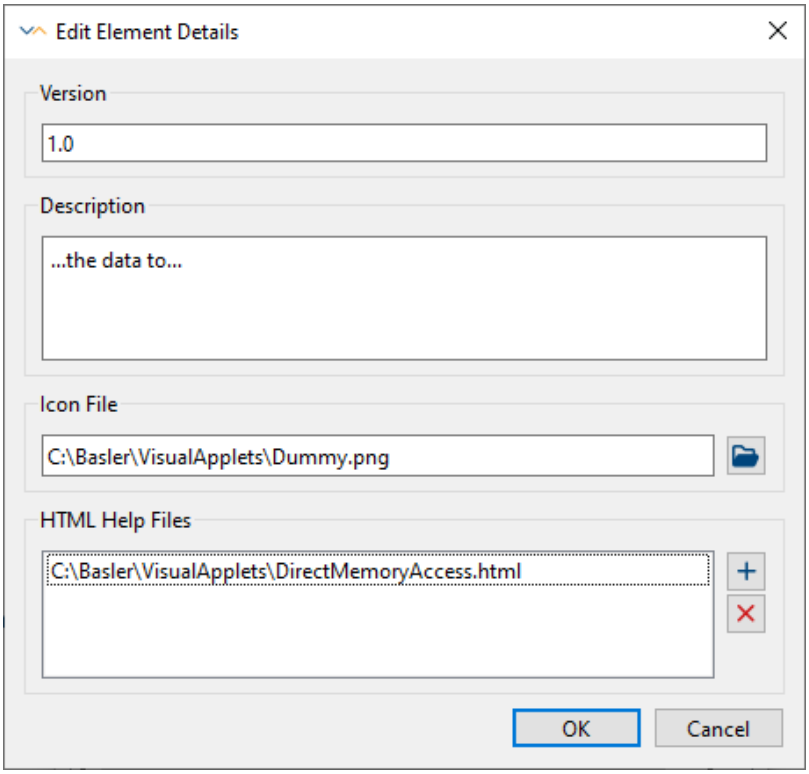



Figure 4.16. Adding documentation, version information, short description, and/or individual GUI Icon

b. Enter a version number for your library element if you want to.

c. Enter a short description for your library element if you want to.

d. To specify a specific icon to be displayed together with the library element on the VisualApplets GUI, in field **Icon File** specify the path to the PNG file or use the navigation button on the right hand side of the field.

e. To add documentation describing your user library element, click the plus button under **HTML Help Files** and navigate to the HTML file that holds the text of your documentation.

You may specify multiple HTML and image files. If you add multiple HTML files, the first HTML file in the list will be the starting page of your documentation. The starting page is displayed as documentation of your user library element (in the information panel, tab **Help**). The additional files may contain images or further HTML files that are linked by the starting page.

**Naming and Storage of Help Files**

When you save your user library element later on, all the files you specify here will be copied to the VisualApplets installation directory, sub-directory **LibraryCache**. The starting page of your documentation (first HTML file in the list) will be renamed and will have the name of the user library element: **<elementname>.html**

To unlink an HTML file from the library element, select the element and click on the delete button. The HTML file will be unlinked, but remains in your file system.

f. Confirm the details specification of your new user library element with **OK**.

11. If you want to protect your user library element, select the *Protection* option. This way, the user library element is made a “black box”. (For details on element protection, see Section 4.2.4, 'Protecting User Library Elements'.)

> **Protected Elements Can't Be Saved to the Legacy File Format**
>
> If you save your user library in the `*.vl` legacy file format, protected elements can't be saved. In this case, either remove the protection or save the user library in the newer and recommended `*.val` file format.
>
> With the legacy file format you can open the user library in VisualApplets version 3.3.2 or older.

12. Click **Save**.

    The optional PNG and HTML files are copied to the VisualApplets installation directory, sub-directory **LibraryCache**.

    If you activated the *Protection* option, an additional dialog *Protect User Library Element* appears.

    a. Make sure protection mode **Password** is activated.

    b. Enter your password and click **OK**.

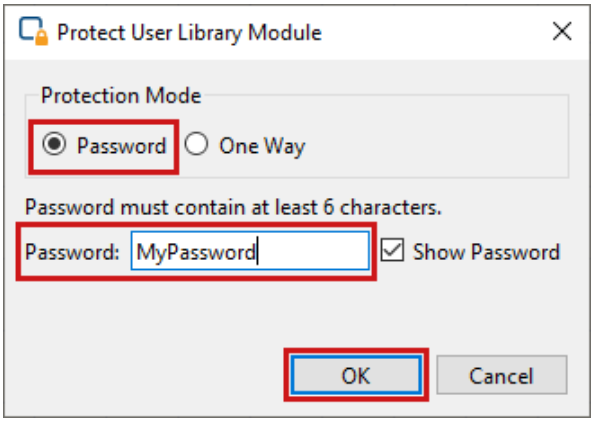



Figure 4.17. Providing a password for a library element

All User libraries and their contents are available in the library panel (see Section 3.3, 'Main Program Window'). Click on the *User Library* tab to display the User Library window.

The icon you selected is visible together with the library element.

Detailed information on an individual element is displayed in the tooltip:

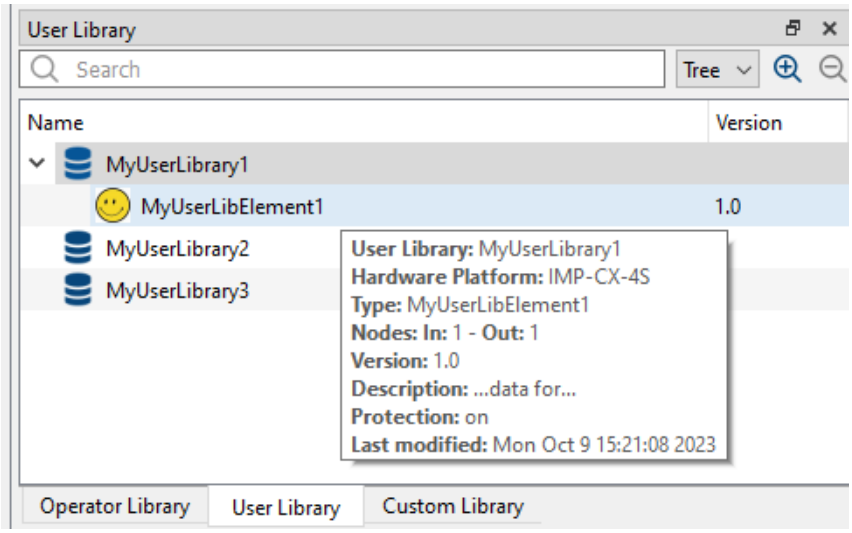



Figure 4.18. Tooltip Information on User Library Element

If you select the new user library element, the according help information is displayed in the information panel under tab **Help**.

13. If you want to proceed working on your design, close the user-library-element design and the user library editor: From the main menu, select **File** -> **Close**.

> **Close element design via File->Close**
>
> It is not enough to close the element design by closing the tab of the design. You need to close the user library editor, too.
>
> To close both, click **File** -> **Close**. Only after closing the design of your user library element (and thus the library editor) in this way, you can re-open the normal design editor and thus your overall applet design.

14. To continue working on your design, open your design.

You can change your user library element any time later on (see section Section 4.2.5, 'Editing User Library Elements'. However, these changes will not be promoted to the instances of the element. You have various options for updating the instances of a user library element if you want to (see section Section 4.2.6, 'Updating Instantiated User Library Elements'.

> **Analysis of User Library Elements**
>
> Design Rules Check Level 1 is possible.
>
> Resource analysis, build, and simulation are not possible.

## 4.2.3. Using the User Library

### 4.2.3.1. Inserting User Library Elements into Your Design

You can create an instance of an element by a simple drag & drop into your design (as you do with operators).

The instantiated module behaves like a hierarchical box, but knows it descends from the user library.

### 4.2.3.2. Adapting Element Instances to Your Design

You can adapt instances of user library elements to fit your overall design. You may need to

- modify individual operator parameters
- modify the layout (adding and deleting of operators and links)



**Changing Link Properties**

Changed link properties do not change the implementation of the library element instance. Example: If a library element has been saved with a parallelism of 4, but its instance is connected to a parallelism of 8, this of course has an influence on the links inside the instance. However, this is not a change of the implementation of the instance as no operator parameters are influenced by a change of the link properties.

After you changed an instance of a user library element - either by modifying individual operator parameters, or by modifying the layout (adding and deleting of operators and links), the instance is no more a mere copy of the user library element.

To keep you informed which instance contains changes, the modified user library element instance is marked:

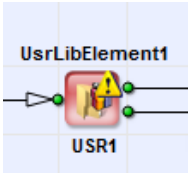



Figure 4.19.

This is very important for you to know, as updating the instances of a user library element (see section Section 4.2.6, 'Updating Instantiated User Library Elements') overwrites the changes you made to an instance. You should always carefully consider the pros and cons before selecting an automatic update for an instance that contains changes.

## 4.2.4. Protecting User Library Elements

If you have purchased an **Expert** license or the **VisualApplets 4** license, you can protect user library elements. After protection has been enabled, the user library element is made a "black box". However, protected elements that contain dynamical parameters or parameters that are referenced from an operator of the *Parameters* library are still visible in the runtime.



**Protected Elements Can't Be Saved to the Legacy File Format**

If you save your user library in the `*.vl` legacy file format, protected elements can't be saved. In this case, either remove the protection or save the user library in the newer and recommended `*.val` file format.

With the legacy file format you can open the user library in VisualApplets version 3.3.2 or older.

There are two ways to protect a user library element:

- Protection via password: The user library element can afterwards be opened and edited via password. Users that do **not** have the password will not even be able to see which operators are used in the user library element (black box).
- Irreversible protection: The user library element is made a black box for ever and cannot be re-opened, not even by yourself.

To protect a user library element via password:

1. Right-click on the element.
2. From the context menu, select **Protect User Library Element**.

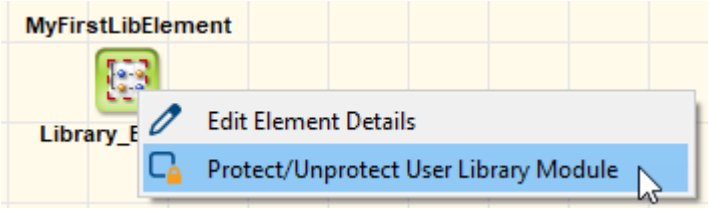



Figure 4.20. Protecting a user library element

3. Make sure protection mode **Password** is activated.
4. Enter your password.

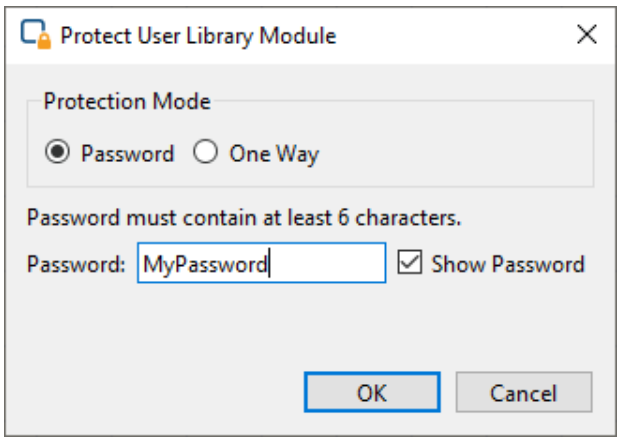



Figure 4.21. Entering password for protected user library element

5. Click **OK**.

**"One-Way" Protection Irreversible**

If you select protection mode *One Way* (instead of *Password*), the user library element can never be re-opened, not even by yourself. If you plan to enhance the element at a later point of time, make sure you select protection mode **Password** instead. Alternatively, you can save a copy of the element (as a hierarchical box or a non-protected operator) before enabling this protection mode.

### 4.2.5. Editing User Library Elements

You can make changes to your user library elements. You can do this by either

- editing the library element itself in an library editor, or by
- overwriting it with a changed instance or a hierarchical box.

However, these changes are not propagated automatically. If you want an instance of a library element to reflect the changes you made to the element, you need to update the instance (see section Section 4.2.6, 'Updating Instantiated User Library Elements').

#### 4.2.5.1. Editing User Library Elements in the User Library Editor

You can edit your user library elements in the user library editor. To open the user library editor:

1. Save the design you are currently working on.
2. In the library pane, go to tab *User Library*.
3. Right-click the element you want to edit.
4. From the context menu of the element, select **Edit**.

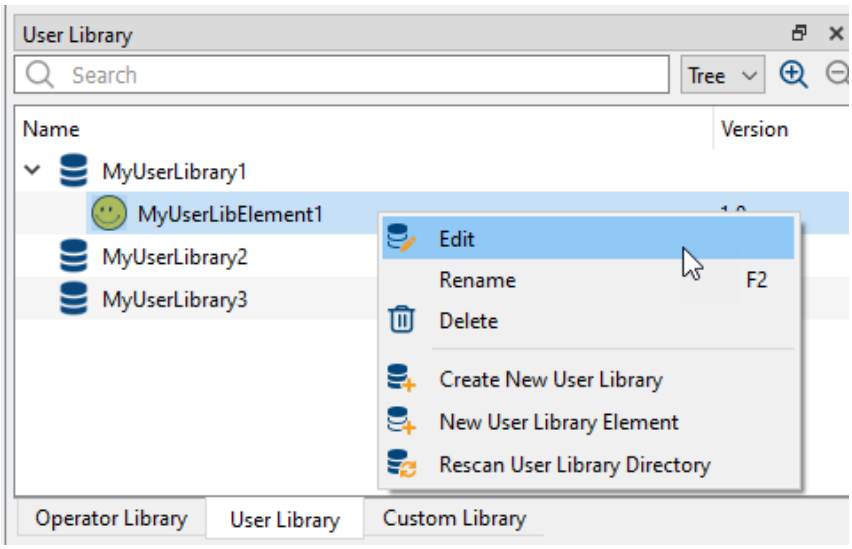



Figure 4.22. Opening the User Library Editor

Now, the design editor closes. Together with the design editor, also the design you have been working on is closed.

The user library editor opens. The user library editor behaves like the normal design editor.

5. Make your changes.
6. Save your changes.

If you want to edit the element description and version number:

7. From the main menu, select **Library** -> **Element Details**.
8. Save the edited element.

**Analysis of User Library Elements:** You can carry out Design Rules Check Level 1. However, bandwidth analysis, resource analysis, build, and simulation are not possible.



**Updating Instances**

When you make changes to user library elements, the instantiated elements in your designs are **not** updated with these changes. If you want an instance of a library element to reflect the changes you made to the element, you need to update the instance (see section Section 4.2.6, 'Updating Instantiated User Library Elements').

### 4.2.5.2. Editing a User Library Element via Overwriting

Alternatively, you can edit a user library element by simply overwriting it. To do so:

1. Save a hierarchical box or an instantiated user library element under the name of an already existing user library element (as described in Section 4.2.2.1, 'Saving a Hierarchical Box as a User Library Element').

**Updating Instances**

When you make changes to user library elements, the instantiated elements in your designs are **not** updated with these changes. If you want an instance of a library element to reflect the changes you made to the element, you need to update the instance (see section Section 4.2.6, 'Updating Instantiated User Library Elements').

In the following sections you get information on how to update instantiated user library elements.

## 4.2.6. Updating Instantiated User Library Elements

In the normal workflow, it happens that user library elements need to be updated. The changes you make to an element, however, have no influence on the instances of this element. To update the instances of an element in your design(s), you can either add your changes manually, or use one of the two update mechanisms provided by VisualApplets (see below).

### 4.2.6.1. Updating Manually

If you decide to update the instances of the changed element manually, you can select multiple instances of the element in your design and change them all simultaneously in one step.

**No Hierarchical Update**

Hierarchical update of user library elements is not supported. If you select more than one module of a hierarchy for updating, only the highest module of the hierarchy is updated.

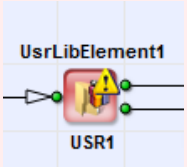
### 4.2.6.2. Replacing Modules via Quick Update

You can update the instances of a user library element in a design via the function Quick Update.

**Updating Overwrites Changes within Instances**

Updating the instances of a user library element overwrites all changes you made to an instance (parameter settings, or adding/deleting operators). You should always carefully consider the pros and cons before selecting an automatic update for an instance that contains changes.

To keep you informed which instance contains changes, modified instances of a library element are marked:




To update the instance of a library element via *Quick Update*:

1. Select the instance you want to update.
2. From the main menu, select **Library** -> **Quick Update from User Library**.

The selected instance of the element is replaced by a new one that reflects the changes you made to the element in the library.

Quick Update is only possible if the instances you want to update have the same number of input and output ports as the edited user library element. It is not possible to use Quick Update for replacing the instance of a user library element type A by an instance of user library element type B.

Via Quick Update, you can update multiple instances of an element at once:

1. Press the **Ctrl** key and hold it pressed while you select all instances you want to update.
2. From the main menu, select **Library** -> **Quick Update from User Library**.

   All selected instances of the element are replaced by new ones.

### 4.2.6.3. Replacing an instance via *Update from User Library*

When you select this option, the instances you want to replace do not have to be of the same type as the library element they are updated with. Thus, you can use this option to replace instances of user library elements or hierarchical boxes by the instance of another user library element. For example, you can replace the instance of a user library element type A by an instance of user library element type B (in contrast to option "Quick Update"). Nevertheless, also **Update from User Library** is only possible if if the instances you want to replace have the same number of input and output ports as the user library element you want them to replace with.

To replace one or more instances of an element by instances of another user library element:

1. Select the modules you want to replace (holding the **Ctrl** key).
2. From the main menu, select **Library** -> **Update from User Library**.

   The dialog window *Update User Library Elements* opens. The left-hand panel shows the module(s) you have chosen for update. The right-hand panel shows your user libraries.

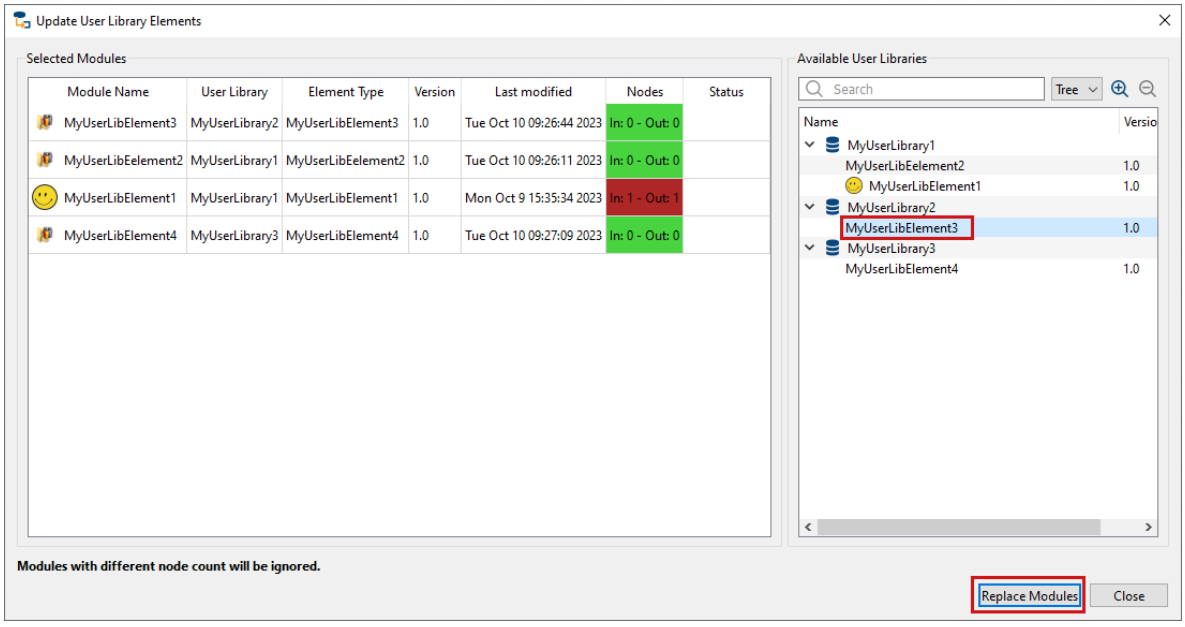



Figure 4.23. Replacement of Instances

3. In the right-hand panel, select the user library element you need. In the left-hand panel, all modules which can be replaced as you desire are highlighted in green (if not, check if the numbers of inputs and outputs fit).
4. Click **Replace Modules**.

   Instantaneously, all listed modules with matching numbers of nodes are replaced by instances of the new user library element.

Example above: The numbers of inputs and outputs do not fit in two cases. Module 28 has only 1 input and 1 output port, whereas USRLibraryELement5 has three inputs and three outputs. Replacement of modules 32, 31, and 33 by USRLibraryELement5 is possible as they have exactly the same number of input and output ports as USRLibraryELement5.

### 4.2.7. Transforming a User Library Module into a Hierarchical Box

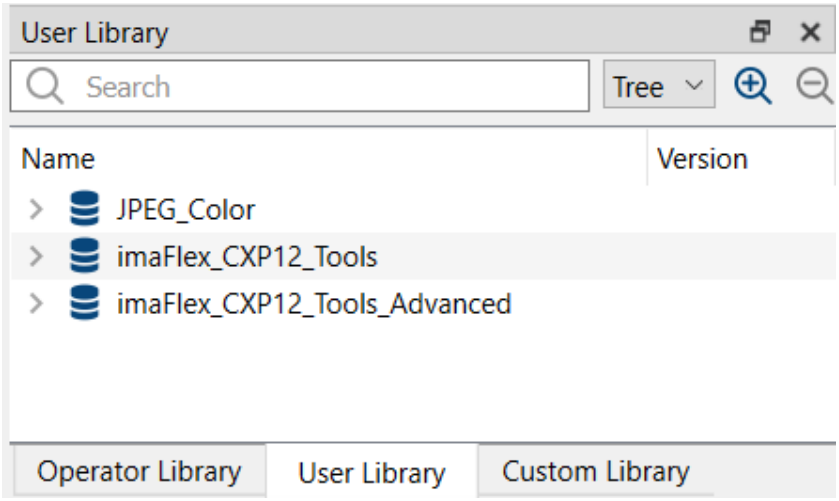
Sometimes you might find it helpful to unlink an element instance from the user library. To do so, you can transform instances of user library elements into hierarchical boxes.

1. Select the module you want to transform in your design. (If you want to transform more than one module, press the **Ctrl** key and select all modules you want to transform.)
2. From the main menu, select **Library** -> **Change to Hierarchical Box**.

Instantaneously, all selected modules are transformed into completely independent hierarchical boxes.

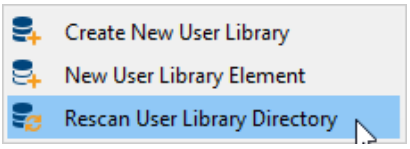
### 4.2.8. Delivered User Libraries

The VisualApplets delivery contains two user libraries:




- **JPEG_Color**
- **imaFlex_CXP12_Tools**
- **imaFlex_CXP12_Tools_Advanced**

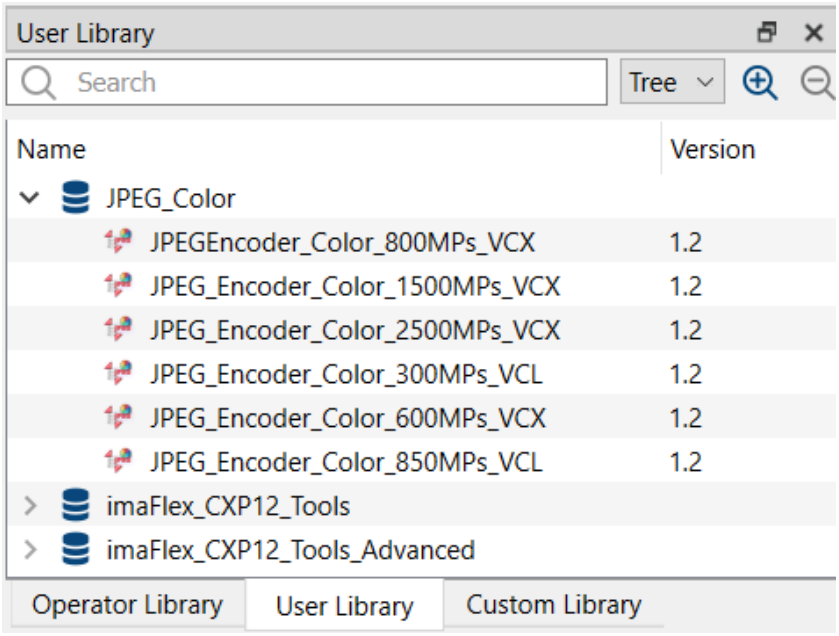
If you don't see these delivered user libraries, right-click into the *User Library* panel, and select Rescan User Library Directory from the context menu that opens up:




VisualApplets then opens the libraries that are located in the directory specified in **System Settings** -> **Path Settings** -> **Libraries/Path for storage of user libraries**.

#### 4.2.8.1. JPEG_Color User Library

The JPEG_Color user library contains the following elements:




**Availability**

The **JPEG_Color** user library is part of **VisualApplets Expert**.

To use elements of the **JPEG_Color** user library, you need either an **Expert** license, a **JPEG Compression Library** license, or the **VisualApplets 4** license.

- **JPEGEncoder_Color_800MPs_VCX**: With this user library element you can convert 8-bit color images into JPEG images. *JPEG_Encoder_Color_800MPs_VCX* is for CXPx2 frame grabbers and provides a bandwidth of 800 MP/s at a clock rate of 160 MHz. The input parallelism is 8.
- **JPEGEncoder_Color_1500MPs_VCX**: With this user library element you can convert 8-bit color images into JPEG images. *JPEG_Encoder_Color_1500MPs_VCX* is for CXPx2 frame grabbers and provides a bandwidth of 1200 MP/s at a clock rate of 160 MHz. The input parallelism is 8.
- **JPEGEncoder_Color_2500MPs_VCX**: With this user library element you can convert 8-bit color images into JPEG images. *JPEG_Encoder_Color_2500MPs_VCX* is for CXPx4 frame grabbers and provides a bandwidth of 2400 MP/s at a clock rate of 160 MHz. The input parallelism is 16.
- **JPEGEncoder_Color_300MPs_VCL**: With this user library element you can convert 8-bit color images into JPEG images. *JPEG_Encoder_Color_300MPs_VCL* is for Camera Link (CL) frame grabbers and provides a bandwidth of 300 MP/s. The input parallelism is 4.
- **JPEGEncoder_Color_600MPs_VCL**: With this user library element you can convert 8-bit color images into JPEG images. *JPEG_Encoder_Color_600MPs_VCL* is for Camera Link (CL) frame grabbers and provides a bandwidth of 600 MP/s at a clock rate of 155 MHz. The input parallelism is 4.
- **JPEGEncoder_Color_850MPs_VCL**: With this user library element you can convert 8-bit color images into JPEG images. *JPEG_Encoder_Color_850MPs_VCL* is for Camera Link (CL) frame grabbers and provides a bandwidth of 850 MP/s at a clock rate of 170 MHz. The input parallelism is 8.

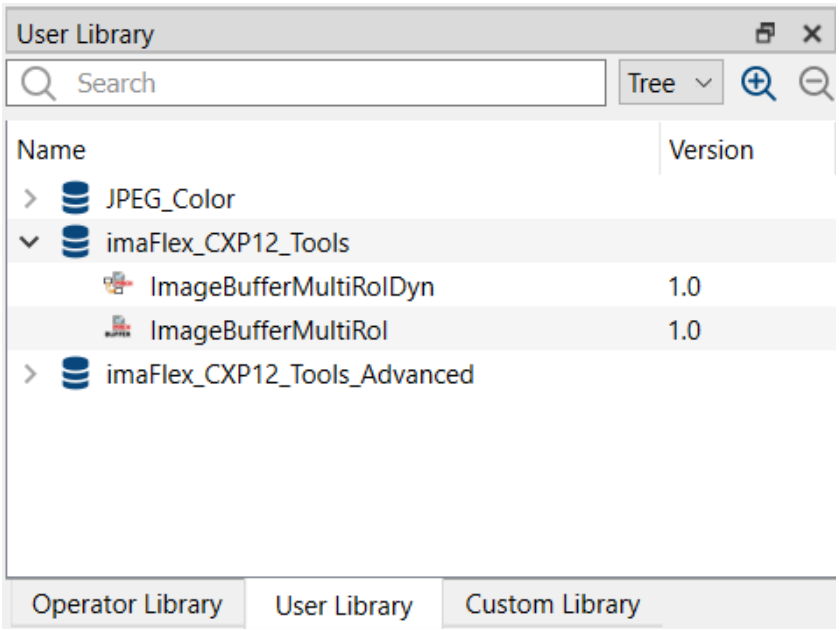
You find detailed documentation for the user library elements in the *Help* panel of your VisualApplets installation.

### 4.2.8.2. imaFlex_CXP12_Tools User Library

**Rather Use the imaFlex_CXP12_Tools_Advanced User Library**

The imaFlex_CXP12_Tools user library is only delivered for backwards compatibility reasons. If possible, use the imaFlex_CXP12_Tools_Advanced user library instead.

The imaFlex_CXP12_Tools user library contains the following elements:




- **ImageBufferMultiRoI**: This operator provides support for multiple regions of interest (ROI) for each buffered image. The user library element *ImageBufferMultiRoI* as tool element for imaFlex platform provides the same functionality as the VisualApplets standard operator *ImageBufferMultiRoI*, which is not supported on imaFlex platform.
- **ImageBufferMultiRoIDyn**: This operator provides support for multiple dynamic regions of interest (ROI) for each buffered image. The user library element *ImageBufferMultiRoIDyn* as tool element for imaFlex platform provides the same functionality as the VisualApplets standard operator *ImageBufferMultiRoIDyn*, which is not supported on imaFlex platform.

### 4.2.8.3. imaFlex_CXP12_Tools_Advanced User Library

The imaFlex_CXP12_Tools_Advanced user library contains the following elements:

- **FrameBufferMultiRoi**: This operator provides support for multiple regions of interest (ROI) for each buffered image. The user library element *FrameBufferMultiRoi* as tool element for imaFlex platforms provides the same functionality as the VisualApplets standard operator *ImageBufferMultiRoi*, which is not supported on the imaFlex CXP-12 platforms.
- **JPEG_Encoder_Color_iF_Penta**: *JPEG_Encoder_Color_iF_Penta* allows you to convert 8-bit color images into JPEG images.

**Availability**

The *JPEG_Encoder_Color_iF_Penta* user library element is part of **VisualApplets Expert**.

To use elements of the *JPEG_Encoder_Color_iF_Penta* user library element, you need either an **Expert** license, a **JPEG Compression Library** license, or the **VisualApplets 4** license.

- **JPEG_Encoder_Color_iF**: *JPEG_Encoder_Color_iF* allows you to convert 8-bit color images into JPEG images.



**Availability**

The *JPEG_Encoder_Color_iF* user library element is part of **VisualApplets Expert**.

To use elements of the *JPEG_Encoder_Color_iF* user library element, you need either an **Expert** license, a **JPEG Compression Library** license, or the **VisualApplets 4** license.

You find detailed documentation for the user library elements in the *Help* panel of your VisualApplets installation. Also, there are examples available that show you how to use the user library elements, see 11. *imaFlex CXP-12 Quad and Penta Implementation Examples*

## 4.3. Custom Operator Libraries

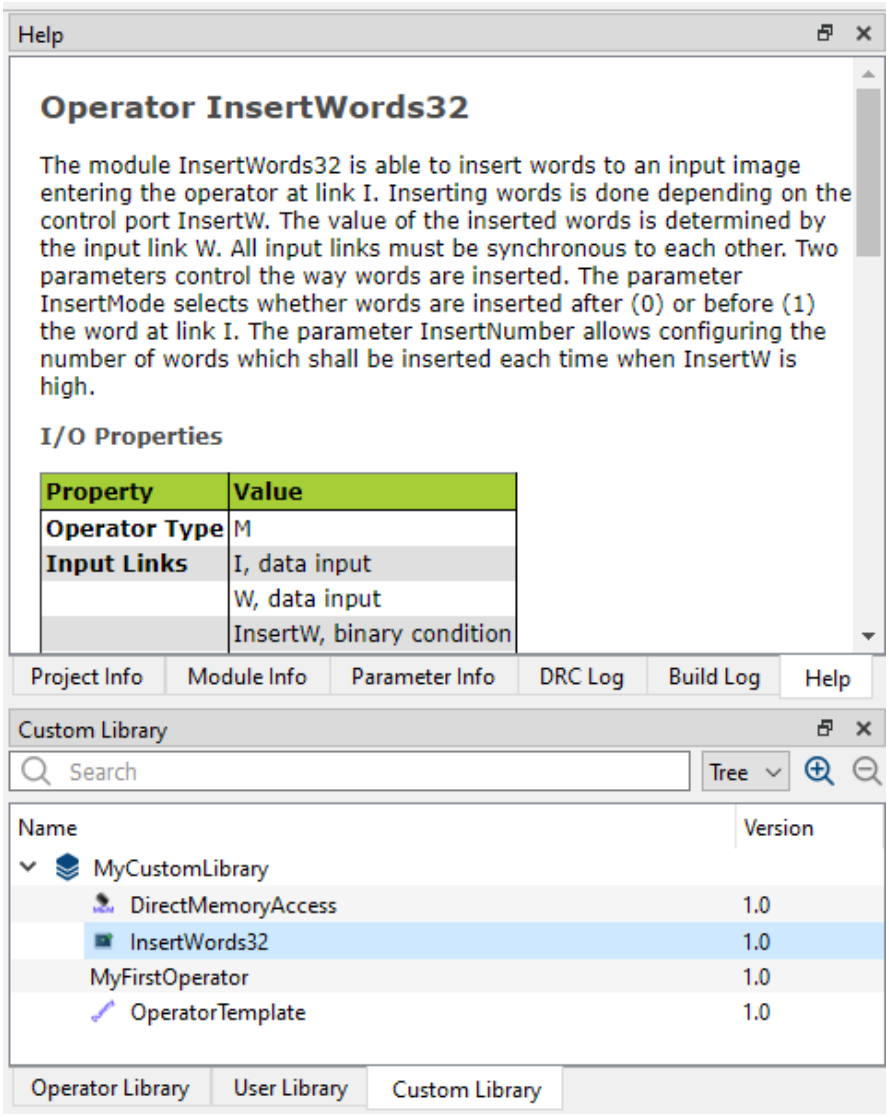
With VisualApplets, you have the possibility to convert image processing modules you have designed in VHDL into VisualApplets operators.

You incorporate your modules as IP cores into VisualApplets. Each IP core builds one operator. After implementation, these operators work like built-in VisualApplets operators. Operators implemented in such a way are called **custom operators**.

**Availability**

The VisualApplets Custom Library feature is part of **VisualApplets Expert**.

To use the Custom Library, you need either an **Expert** license or the **VisualApplets 4** license.




To make your custom operators available on the VisualApplets GUI, you also need to define one or more custom libraries that contain the custom operator(s.) Each custom operator needs to be part of one specific custom library.

### 4.3.1. Workflow

You add a new **custom operator** to VisualApplets in just a few steps. You can complete the whole work flow by your own:

1. Specify the custom operator's main properties and its interface directly on the VA GUI (operator name, operator version, number and properties of required image in, image out, memory ports, etc.).
2. Based on your input of step 1, let VisualApplets generate the VHDL code for the operator interface (black box) and a VHDL test bench for testing your implementation.
3. Wrap your HDL code so that its interface matches the generated black box. For testing your implementation the automatically generated test bench may help.
4. Create a net list of your implementation. Also create a constraints file if required.
5. Optionally, create the operator documentation (for the operator help window) and a simulation model (that later allows to simulate a VA design containing the custom operator).
6. Edit the custom operator in VA again: Add the generated netlist and optionally also the help files and simulation model.

After these steps, your image processing module is available as custom operator directly in VisualApplets and can be used the same way as any other operator. The custom libraries are saved as `*.val` or `*.vl` files (similar to the user libraries). They can be deployed and distributed in this format.

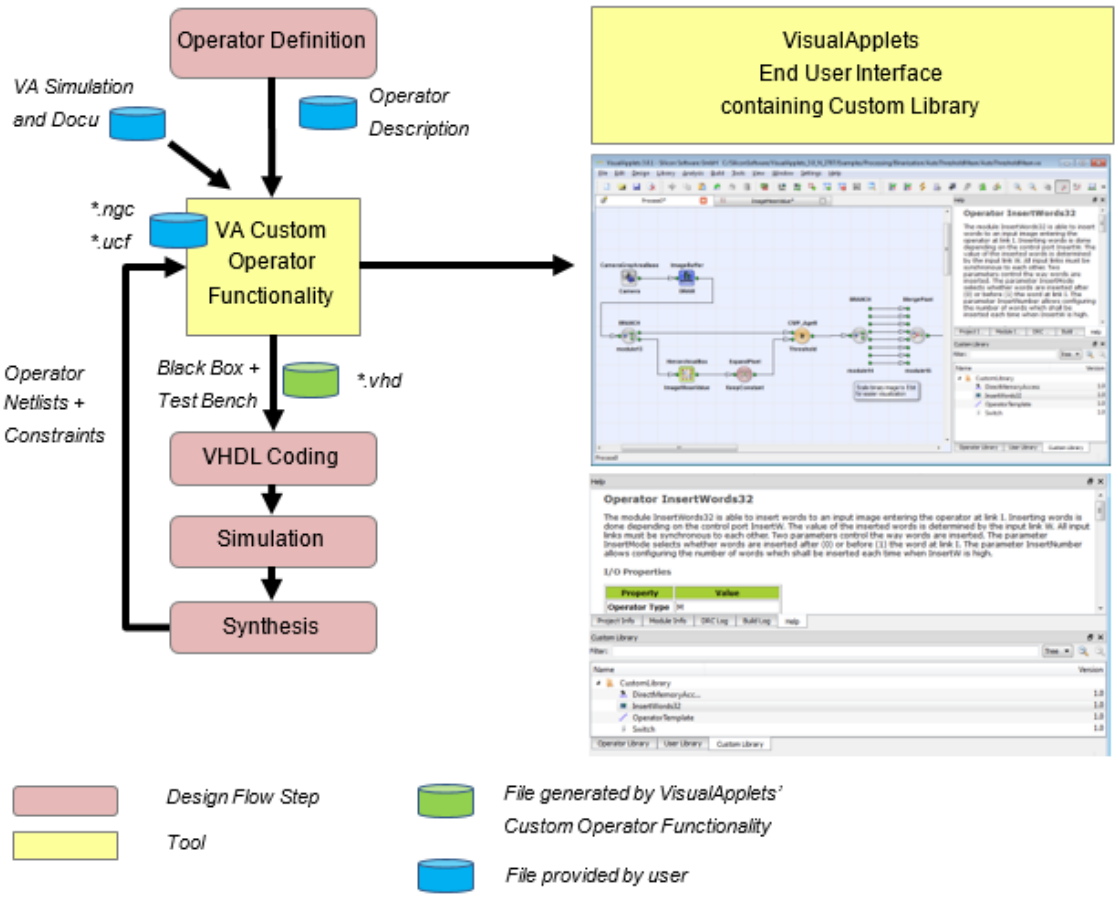



**Figure 1: Custom Operators design flow.**

### 4.3.2. VisualApplets Custom Operator Functionality

VisualApplets (i.e., the VisualApplets Custom Operator Functionality) is used two times during this work flow:

1. For the generation of an operator prototype in VisualApplets allowing to export HDL code for defining the concerning IP core interface (black box and test bench).

2. For completing the operator by adding the necessary files for synthesis and (optionally) simulation and help content.

**Generation of Operator Prototype:** The VA Custom Operator Functionality lets you create an operator prototype which can immediately be used for instantiating the operator in Visual Applets. For this operator prototype a black box interface and an RTL level simulation entity for emulating the communication ports of the generated operator interface can be exported. Then you can start coding (i.e., implementing your HDL code complying with the interface of the black box) and simulating your custom operator design. The resulting FPGA design you then synthesize to an EDIF or NGC netlist. Optionally, you add a constraints file, create a dynamic link library for VisualApplets high-level simulation, and write HTML documentation for the VisualApplets GUI.

**Completing the operator definition:** The VA Custom Operator Functionality lets you specify the netlist, simulation library and documentation files. Supplemented with these files the operator is ready for use immediately.

### 4.3.3. Operator Types

VisualApplets knows different types of operators and ports, depending on the underlying flow control mechanism. Operators may be of type O or type M.



**Custom Operator Type: M**

Custom operators are always of type M.

### 4.3.4. Synchronous and Asynchronous Operator Ports

Operator ports can be synchronous or asynchronous. Being synchronous in VisualApplets basically means that data of several ports is transferred synchronously, whereas ports which are asynchronous to each other support non-aligned communication patterns.

Ports are only synchronous if they have a common M-source, or if they are sourced from a SYNC module; any constellation of O-operators may be between that source and the ports.

Depending on the relation of the operator input ports to each other, we differentiate between the following options:

1. **Synchronous inputs**: All input ports are synchronous to each other. There is one output port.

2. **Asynchronous inputs**: Some of the input ports are asynchronous to each other and all outputs are synchronous to each other.

Operators with asynchronous outputs are not allowed. Operators with synchronous inputs may only have a single output. If more than one output is required, the inputs must be declared as being asynchronous.



**Defining Multiple Outputs**

If you want to create an operator with multiple outputs, you need to declare its inputs to be asynchronous. Multiple outputs are always synchronous.

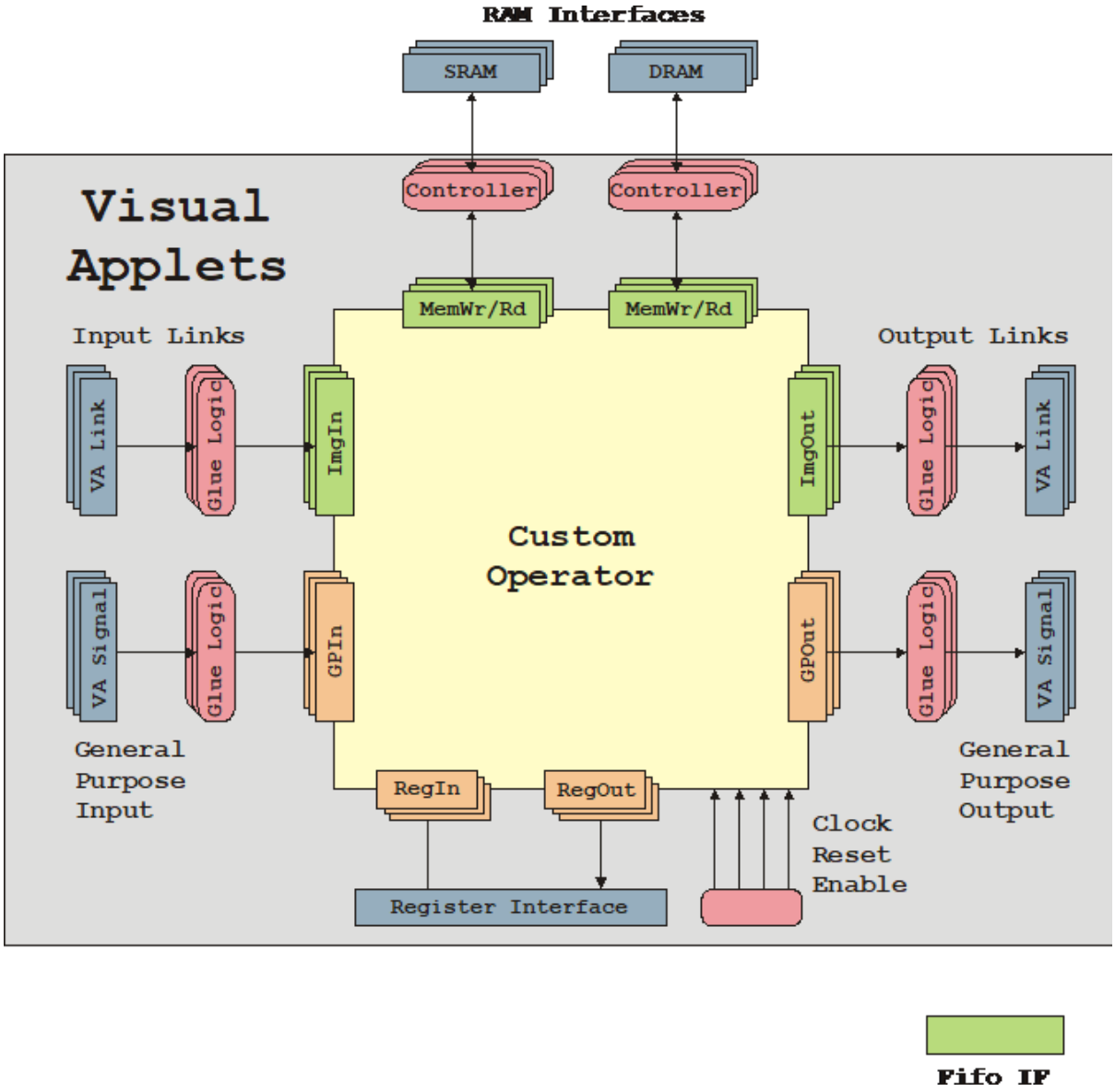
Examples for both classes (built-in Visual-Applets operators):

| | | |
|---|---|---|
|  | *RemoveImage* | M-Operator with synchronous inputs and one output. |
|  | *SYNC* | M-Operator with asynchronous inputs and multiple, synchronous outputs. |

### 4.3.5. Interface Architecture

VisualApplets custom operator interfaces are designed for smoothly integrating your new operators so they behave inside VisualApplets like built-in operators. You can define any number of input and output ports for your custom operators.

- **Image In / Image Out:** The *Image In* and *Image Out* ports may support multiple image formats. They are driven by simple-to-use FIFO interfaces. The FIFOs reside in the VA part of the custom operator, so that you only need to implement a flow control, but not the FIFO.
- **Memory ports:** You also can define any number of memory ports. They also use FIFOs residing in the VA part of the custom operator.
- **GPIO ports:** In addition to the image ports, you can define general-purpose I/O ports, e.g., for communicating asynchronous signals to the operator.
- **Registers:** To allow the final user of your operator to configure the operator and to get access to status information, you can define any number of write and read registers.
- **Clock:** The ports for receiving clock pulses are set up automatically for every custom operator.
- **Reset/Enable port:** The ports for receiving reset or enable commands are set up automatically for every custom operator.




The following sections describe the different types of interfaces shown in the above figure in detail.

### 4.3.5.1. Clock Interface

VisualApplets connects two clock inputs – the design clock and a second clock synchronous to the design clock but with double frequency. All interfaces except the memory interface must be synchronous to the design clock. The memory interface may be configured using the design clock or the double frequency clock for the read and/or write interface.

### 4.3.5.2. Reset and Enable

The Reset and Enable inputs are driven by the according "process enable" and "process reset" signals of the VA-process where the operator is instantiated. Make sure you implement the following behavior as reaction to these signals into your operator:

- Assertion of Reset puts the operator in its init state.
- Assertion of Enable starts processing.
- Deactivating Enable stops processing.

- (When Enable=0, the output FIFOs of the operator are not read. Depending on the state of the image processing pipeline some data may still be written to the input ports but the flow control safely prevents that any FIFO content gets corrupted.)
- Reset is only asserted when Enable=0

The following behavior to these signals is implemented in the VA part of the custom operator:

- Reset will empty all port interface FIFOs.
- Reset and Enable have no effect on the parameters of the operator.
- Reset and Enable have no effect on the GPIO interface of the operator.

### 4.3.5.3. Register Interface

For communicating operator parameters and status, the custom operator may be supplied with an arbitrary number of VisualApplets parameters. Each of the parameters translates to a separate register port of the custom operator. VisualApplets cares for dispatching the accesses to and from the operator registers.

### 4.3.5.4. Interfaces for Image Data

#### 4.3.5.4.1. Image Protocols

You can define the image protocols that will be supported by the image in and image out ports of your custom operator. The future user of your operator will then be able to select from the list of image protocols you provide.

VisualApplets offers the following image formats to be supported by your operator's **ImgIn** and **ImgOut** ports:

- **grayXxP**: gray image with **X** bits per pixel and parallelism **P**
- **rgbYxP:** color image with **Y**/3 bits per color component (red, green, blue) and parallelism **P**
- **hsiYxP:** color image with **Y**/3 bits per color component (HSI color model) and parallelism **P**
- **hslYxP:** color image with **Y**/3 bits per color component (HSL color model) and parallelism **P**
- **hsvYxP:** color image with **Y**/3 bits per color component (HSV color model) and parallelism **P**
- **yuvYxP:** color image with **Y**/3 bits per color component (YUV color model) and parallelism **P**
- **ycrcbYxP:** color image with **Y**/3 bits per color component (YCrCb color model) and parallelism **P**
- **labYxP:** color image with **Y**/3 bits per color component (LAB color model) and parallelism **P**
- **xyzYxP:** color image with **Y**/3 bits per color component (XYZ color model) and parallelism **P**

Additionally, the image dimension and the information whether pixel components are signed or unsigned can be coded by optional suffixes.

The pixel data width **X** is limited to 64 bit. The width **Y** must be a multiple of 3 and is limited to 63 bit. The parallelism **P** defines the number of pixels which are contained in a single data word at the interface port. It must be chosen from following set of allowed values: **P** = {1, 2, 4, 8, 16, 32, 64}.

Packing of image data into words of a given interface width **N** must follow certain rules:

- The data of all **P** pixels must fit in a single word of length **N**. The data is stored LSB aligned which means that for a pixel width **Z** (**Z**=**X** for grey, **Z**=**Y** for color) data is distributed as follows: Pixel[0]->Bits[0..**Z**-1] .. Pixel[**P**-1]->Bits[(**P**-1)*__Z__..**P***__Z__-1].
- For RGB images the three color components are packed LSB aligned into a sub word [0..Y-1] in the following order: red uses the bits [0..**Y**/3-1 ], green the bits [Y/3..2*Y/3-1] and blue the bits [2*Y/3..3*Y/3-1].

- For YUV color images the same rules than for RGB applies where Y takes the role of red, U that of green and V the role of blue.
- For HSI color images the same rules than for RGB applies where H takes the role of red, S that of green and I the role of blue.
- For LAB color images the same rules than for RGB applies where L takes the role of red, A that of green and B the role of blue.
- For XYZ color images the same rules than for RGB applies where X takes the role of red, Y that of green and Z the role of blue.

In VisualApplets, any link carries the properties maximum image width and maximum image height. VisualApplets lets you define optional constraints for the maximum width and height for any of the supported image protocols of the custom operator separately.

For an image interface port, you define a list of allowed image protocols. This list makes up a subset of the possible VisualApplets image formats (see above). A format can be described by the following properties:

- Data type uint or int
- Pixel data bit width N = [1..64]
- Gray or color format (single or three data components with aggregated width N)
- Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
- 2D, 1D, or 0D
- Parallelism P = {1,2,4,8,16,32,64}

Implicitly it is assumed that the kernel size is 1x1. The listed formats are numbered starting from zero and therewith define an ID.

When working with the final operator in VisualApplets, the user of your operator can select any of the formats you list here for the image communication port in question. According to the selection made by the VA user, the corresponding ID will be output to the related custom operator port.

This enables the custom operator to adapt its behavior to the selected format.

### 4.3.5.4.2. Image Input Ports

Image input ports allow to communicate image data from the VisualApplets process to the custom operator. These ports are named **ImgIn**. If you designed the custom operator to support configuration of its input channel(s) (see section Section 4.3.5.4.1, 'Image Protocols'), several different protocols can be driven through a single port selected by the corresponding format parameter within VisualApplets. The interface basically consists of a FIFO and a parameter register providing an ID for the actually used data format. The custom operator must care for reading the FIFO and interpreting the image data according to the protocol of the selected image format. The operator must guarantee a correct flow control according to the status pins providing information about the filling state of the FIFO, i.e., no data may be read when the FIFO is empty.

For an image interface port, a list of allowed image formats needs to be defined. This list makes up a subset of possible VisualApplets image formats (see section Section 4.3.5.4.1, 'Image Protocols') where a format can be described by the following properties:

- Data type uint or int
- Pixel data bit width N = [1..64]
- Gray or color format (single or three data components with aggregated width N)
- Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
- 2D, 1D, or 0D
- Parallelism P = {1,2,4,8,16,32,64}

Implicitly it is assumed that the kernel size is 1x1. The listed formats are numbered starting from zero and therewith define an ID.

When working with the final operator in VisualApplets, the user can select any of the formats you list here for the concerning image communication port. According to the selection made by the VA user, the corresponding ID will be output to the related Custom Operator port. This enables the custom operator to adapt its behavior to the selected format.

#### 4.3.5.4.3. Image Output Ports

Image output ports allow communicating image data from the custom operator to the VisualApplets process. These ports are named **ImgOut**. If you designed the custom operator to support appropriate configuration of its output channel(s) (see section Section 4.3.5.4.1, 'Image Protocols'), several different protocols can be driven through a single port selected by the corresponding format parameter within VisualApplets. The interface basically consists of a FIFO and a parameter register providing an ID for the actually used data format. The custom operator must care for feeding the FIFO with image data according to the protocol of the selected image format. The operator must guarantee a correct flow control according to the status pins providing information about the filling state of the FIFO, i.e., no data may be written when the FIFO is full.

For an image interface port, a list of allowed image formats needs to be defined. This list makes up a subset of possible VisualApplets image formats (see section Section 4.3.5.4.1, 'Image Protocols') where a format can be described by the following properties:

- Data type uint or int
- Pixel data bit width N = [1..64]
- Gray or color format (single or three data components with aggregated width N)
- Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
- 2D, 1D, or 0D
- Parallelism P = {1,2,4,8,16,32,64}

Implicitly it is assumed that the kernel size is 1x1. The listed formats are numbered starting from zero and therewith define an ID.

When working with the final operator in VisualApplets, the user can select any of the formats you list here for the concerning image communication port. According to the selection made by the VA user, the corresponding ID will be output to the related custom operator port. This enables the custom operator to adapt its behavior to the selected format.

### 4.3.5.5. General purpose I/O

The General Purpose I/O interface allows connecting dedicated signal pins of the custom operator. Every GPIO port maps to a pin of the custom operator which is either an input or an output.

Bidirectional pins are not supported. In VisualApplets, the corresponding operator ports are of type SIGNAL.



**Bidirectional Pins not Supported**

The GPIO pins must be either an input or an output. Bidirectional pins are not supported.

### 4.3.5.6. Memory Interface

A custom operator may be set up for accessing one or more banks of memory. The concerning memory ports have a FIFO like interface for write and read commands. The FIFOs reside in the VA part of the custom operator, so that you only need to implement a flow control, but not the FIFO. The timing of forwarding the FIFO content to the memory controller attached to the custom operator is fully controlled by VisualApplets.

### 4.3.6. Defining an Individual Custom Operator via GUI

First of all, you need to enter some details describing your new custom operator.

VisualApplets uses these details for generating a VHDL black box for your custom operator and an according test bench for simulation.

You enter the configuration for your individual custom operator via the VisualApplets GUI. VisualApplets makes the specified operator available for use in a design immediately, even if the operator specification is incomplete concerning netlist, simulation model and documentation.



**Custom Library File**

A custom library with all contained operators is stored as one single `<LibaryName>.val` or `<LibaryName>.vl` file. `<LibaryName>` is the name of the custom library.

This file can be distributed and directly applied in VisualApplets. It simply needs to be copied into the Custom Library directory which is specified in the VisualApplets settings.



**Operator Configuration in XML Format**

VisualApplets stores the custom operator specification in XML format. You can export the XML content from the custom library to a file, e.g., for handling it in a version control system. On the other hand you can import the XML for adding a custom operator (see section Section 4.3.15.3, 'Importing and Exporting Individual Custom Operators'). You do not need to know how this XML file looks like. However, if you want to have a look, refer to section Section 4.3.17, 'XML Format for Custom Operator Specification'.

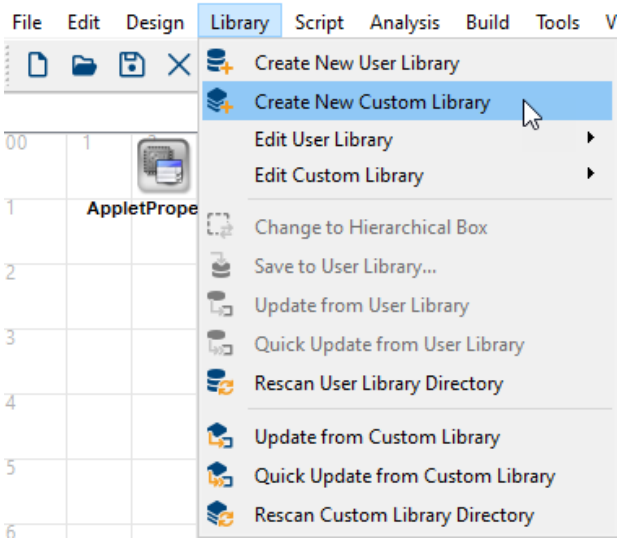
#### 4.3.6.1. Creating a New Custom Library

Before you can start to define a new custom operator, you need to create a custom library where the new operator belongs to.

If you already have a custom library available where the new custom operator will belong to, skip this section and proceed with section Section 4.3.6.2, 'Creating a New Custom Operator'.

To create a new custom library:

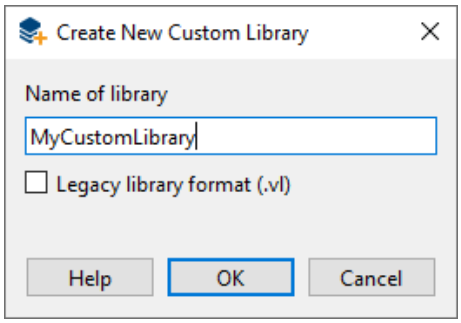
1. In menu **Library**, select menu item **Create New Custom Library**.




2. Give a name to your new custom library.

3. By default, the new custom operator library is saved in the `*.val` file format. If you want to open this library in VisualApplet version 3.3.2 or older, select **Legacy library format (.vl)**.

4. Click **OK**

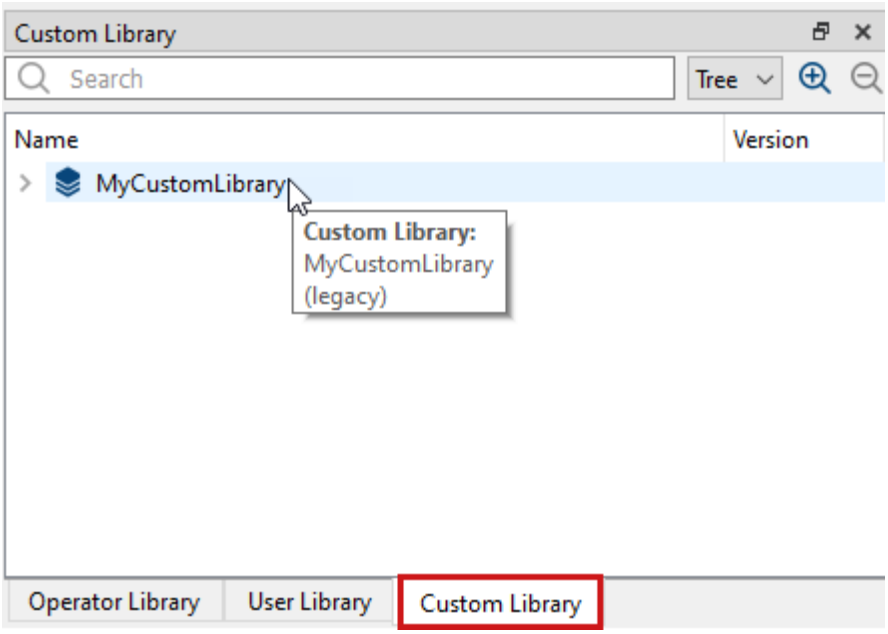



**Comply with Conventions for Valid C Identifiers**

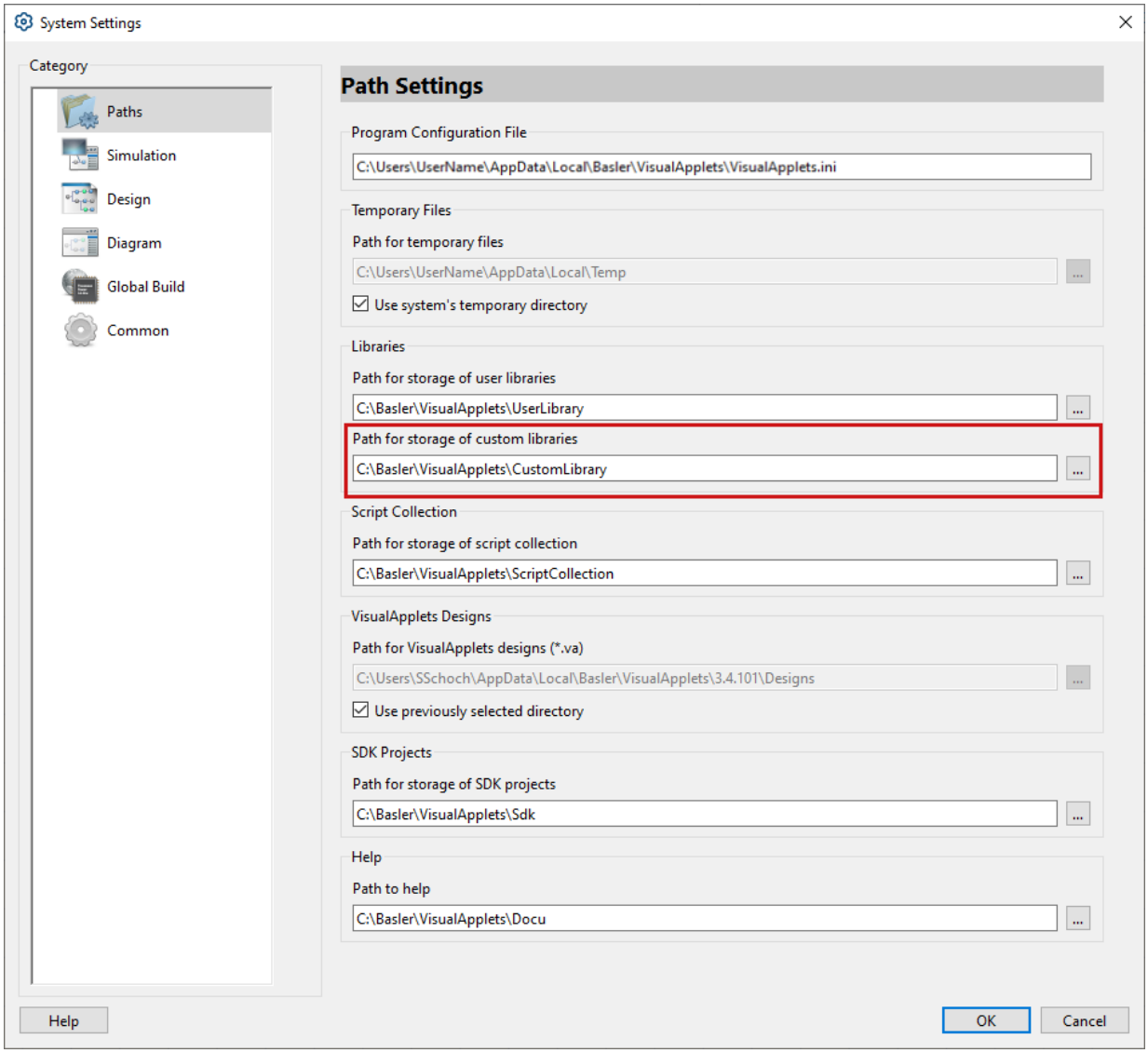
When defining the library name, make sure you adhere to the conventions for valid C identifiers.

Now, the new custom library is created. You can see it in the operator panel under the **Custom Library** tab:

When you move the mouse over the custom library name, a tooltip shows whether the library is saved in the `*.vl` legacy file format:



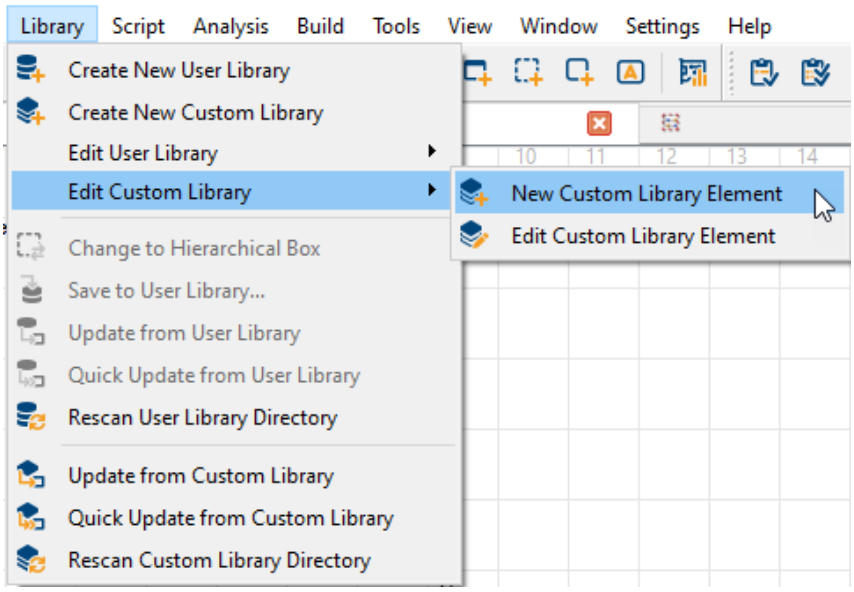

For creating a new custom library, you may need to specify a directory where all custom-library-related files are stored. You do this under **Settings** -> **System Settings** -> **Paths**.



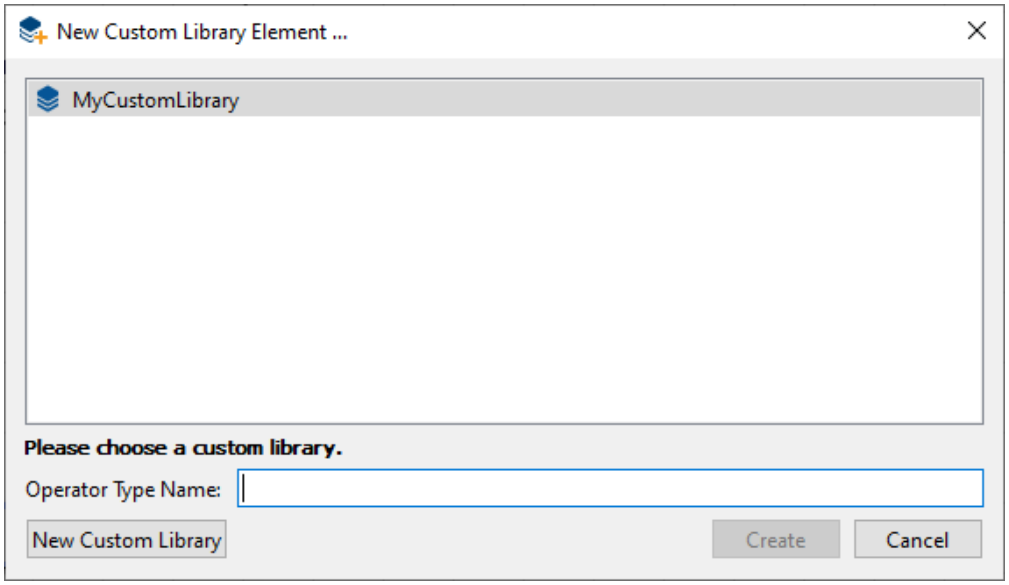

### 4.3.6.2. Creating a New Custom Operator

To define a new custom operator:

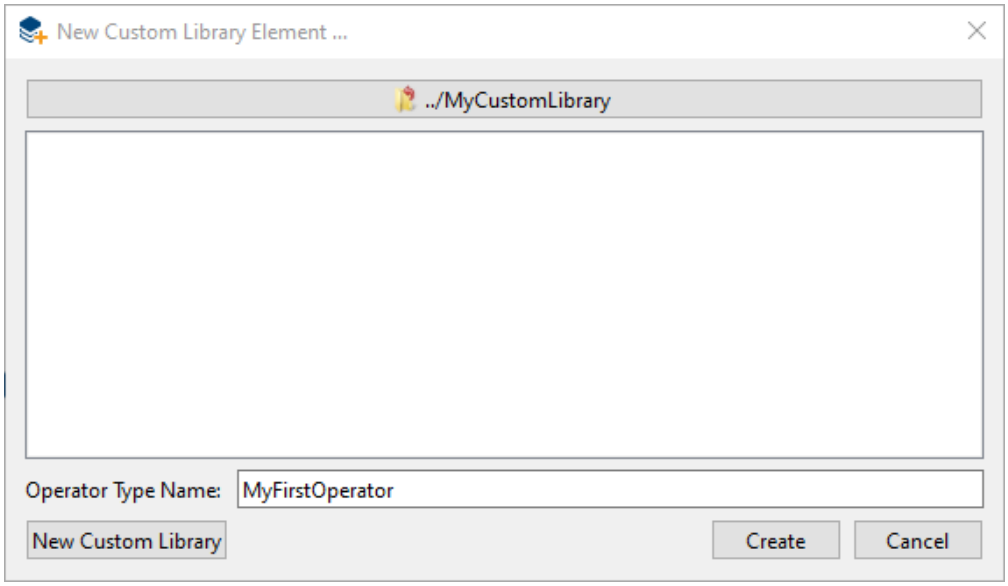
1. In menu **Library**, select menu item **Edit Custom Library**.
2. In the submenu that opens, select **New Custom Library Element**.




3. In the window that opens, select a custom library via double-click on the library name.




4. Enter a name for your custom operator:



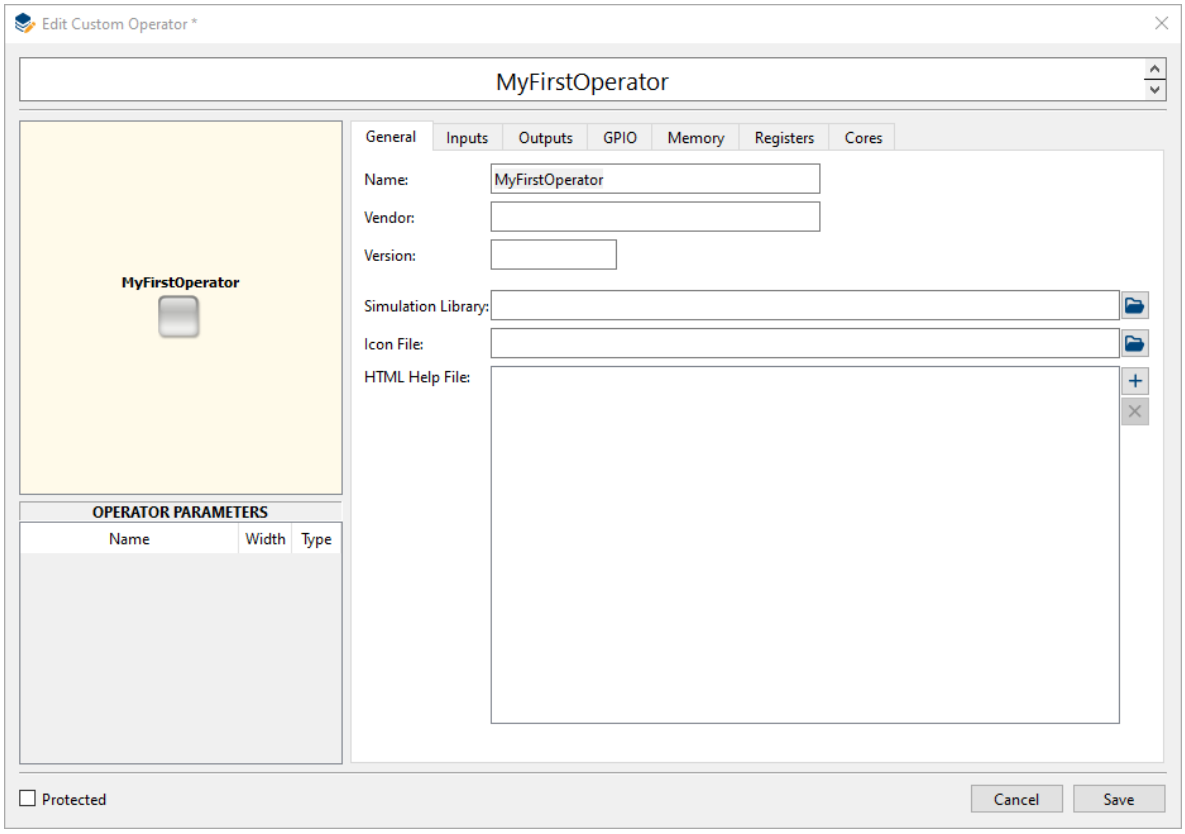

**Comply with VHDL Naming Conventions**

When defining the operator name in the VA GUI, make sure you conform to the VHDL naming conventions.

VHDL valid names are defined as follows:

"A valid name for a port, signal, variable, entity name, architecture body, or similar object consists of a letter followed by any number of letters or numbers, without space. A valid name is also called a named identifier. VHDL is not case sensitive. However, an underscore may be used within a name, but may not begin or end the name. Two consecutive underscores are not permitted."

5. Before saving, you must define the properties of your new operator:



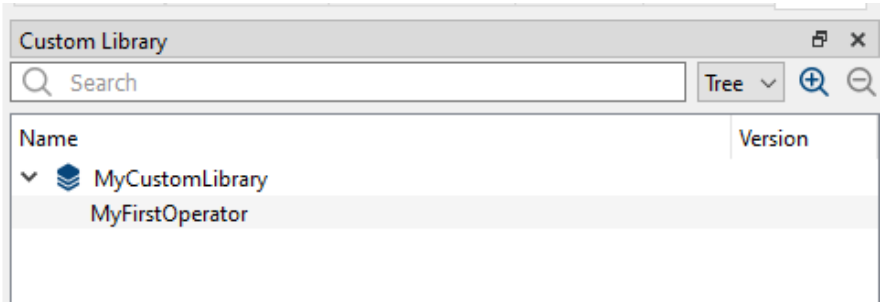

For detailed information about defining the properties of your new custom operator, see Section 4.3.6.3, 'Defining Basic Information about Custom Operator'.

6. Click the **Create** button.

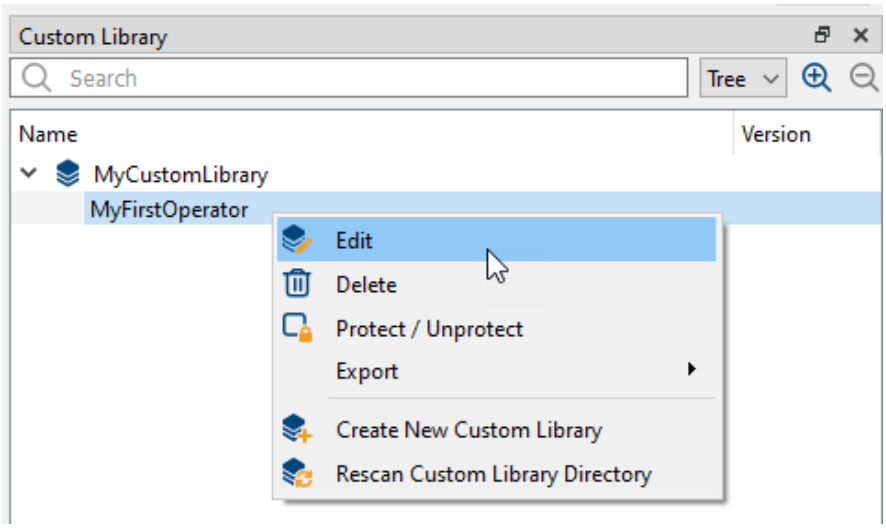
   The dialog **Edit Custom Operator** opens. Here, you can define your custom operator.

7. Click the **Save** button.

Now, your new custom operator is visible under the custom library it belongs to:




Once you have created a new custom operator and saved it to VisualApplets, you can interrupt your work and proceed any time. To proceed, you go to the **Custom Library** tab, open the library, right-click on the operator name, and from the sub menu, select **Edit**.




**Use Operator Template Instead**

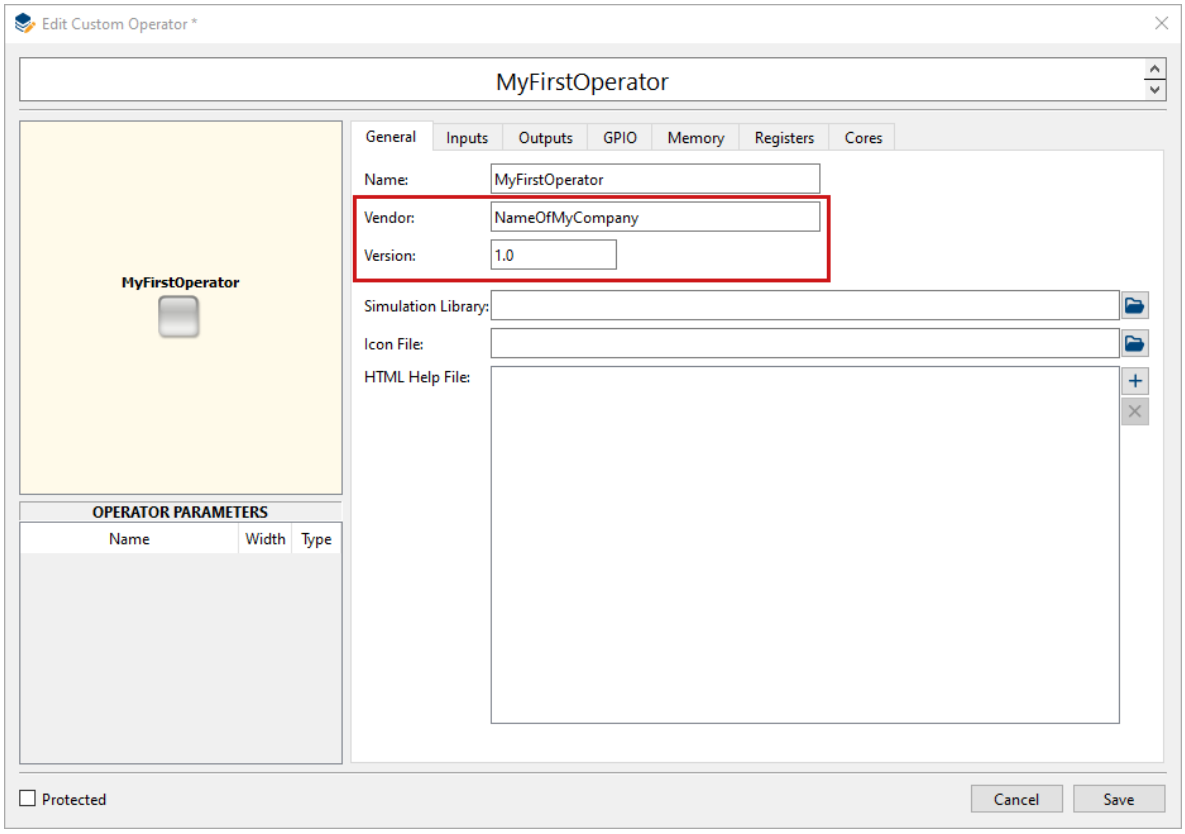
Alternatively, you can use the custom operator template provided in your VisualApplets installation to define new custom operators. How to use the template, see section Section 4.3.16.2, 'Custom Operator Template'.

### 4.3.6.3. Defining Basic Information about Custom Operator

In a first step, you define your custom operator's interface.

1. Provide your vendor name. You can enter any string. This information is intended for operator identification by the user.

2. Provide a version number for your operator, e.g., version *1.0*. You can enter any number but you should comply with the version scheme *<major>.<minor>*. This information is intended for operator version identification by the user.

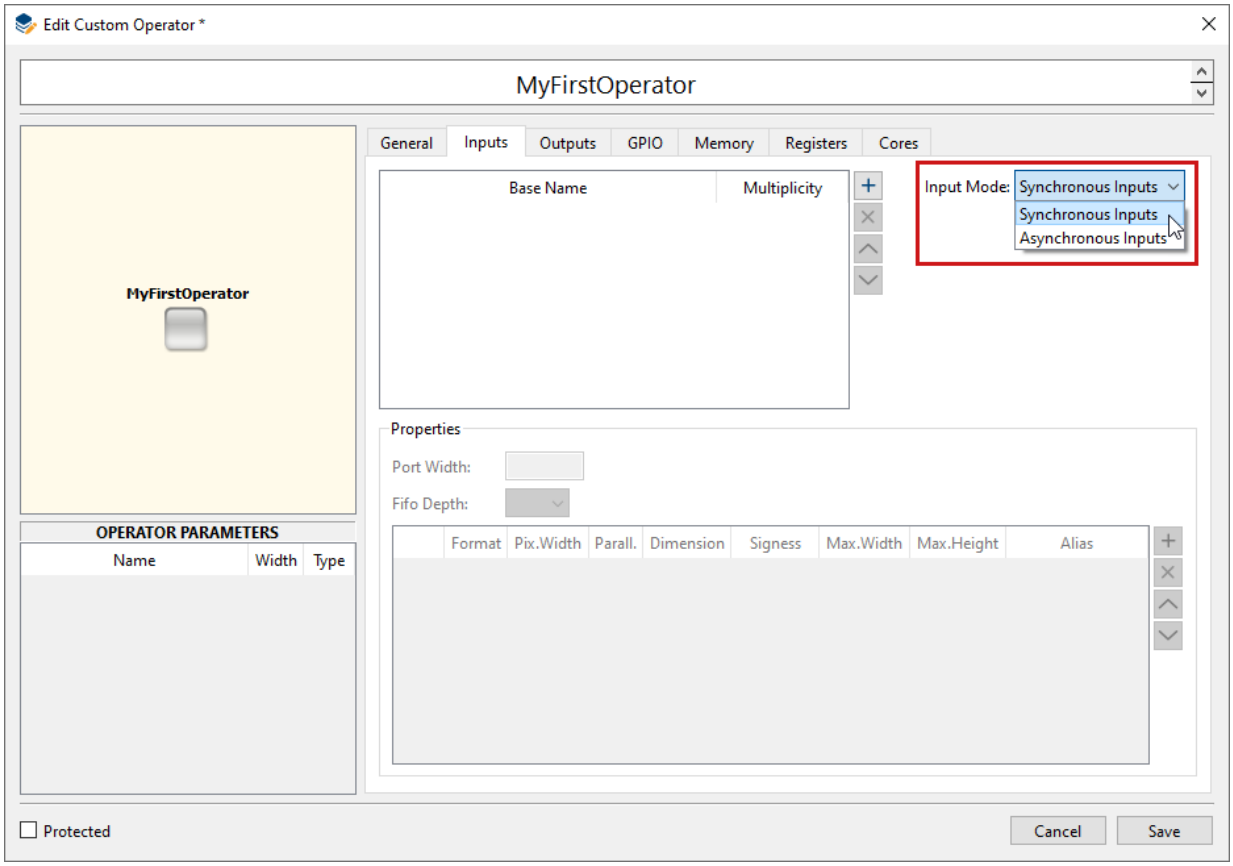



3. Proceed to the tab **Inputs**.

### 4.3.6.4. Defining the Image Input Ports

Under tab **Inputs**, you describe the properties of the image input ports.

1. First of all, you define the input mode of your custom operator's **ImgIn** ports:




## Synchronous and Asynchronous Operator Ports

Operator ports can be synchronous or asynchronous. Being synchronous in VisualApplets basically means that data of several ports is transferred synchronously, whereas ports which are asynchronous to each other support non-aligned communication patterns.

Ports are only synchronous if they have a common M-source, or if they are sourced from a SYNC module; any constellation of O-operators may be between that source and the ports.

Depending on the relation of the operator input ports to each other, we differentiate between the following options:

**Synchronous inputs:** All input ports are synchronous to each other. There is one output port.

A**synchronous inputs:** Some of the input ports are asynchronous to each other and all outputs are synchronous to each other.

Operators with asynchronous outputs are not allowed. Operators with synchronous inputs may only have a single output. If more than one output is required, the inputs must be declared as being asynchronous.

If you want to create an operator with multiple outputs, you need to declare its inputs to be asynchronous. Multiple outputs are always synchronous.

Examples for both classes (built-in VisualApplets operators):

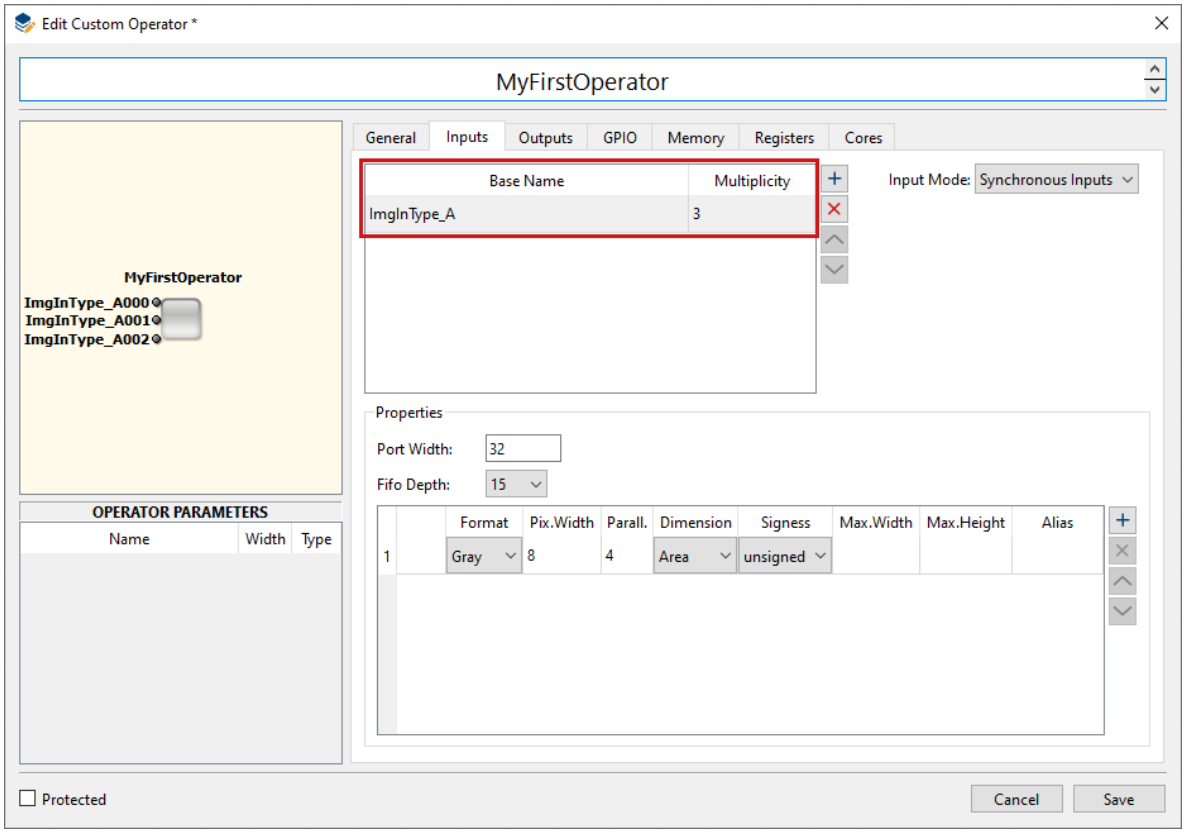
| | | |
|---|---|---|
| | *RemoveImage* | M-Operator with synchronous inputs and one output |

| | | |
|---|---|---|
| | *SYNC* | M-Operator with asynchronous inputs and multiple, synchronous outputs |

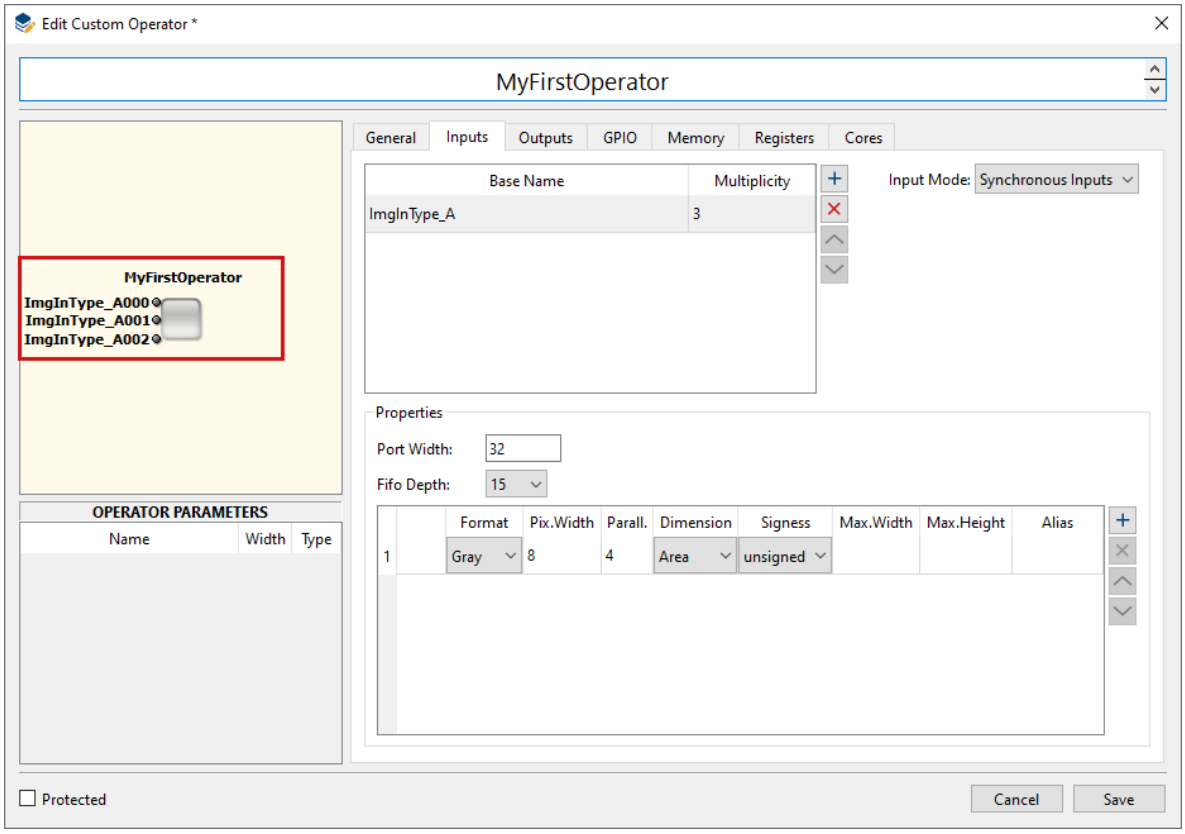
You can define one or more image input ports (**ImgIn**). Each **ImgIn** port may be used as often as you specify.

1. Click on the **Plus** button + to create a first image in (**ImgIn**) port.

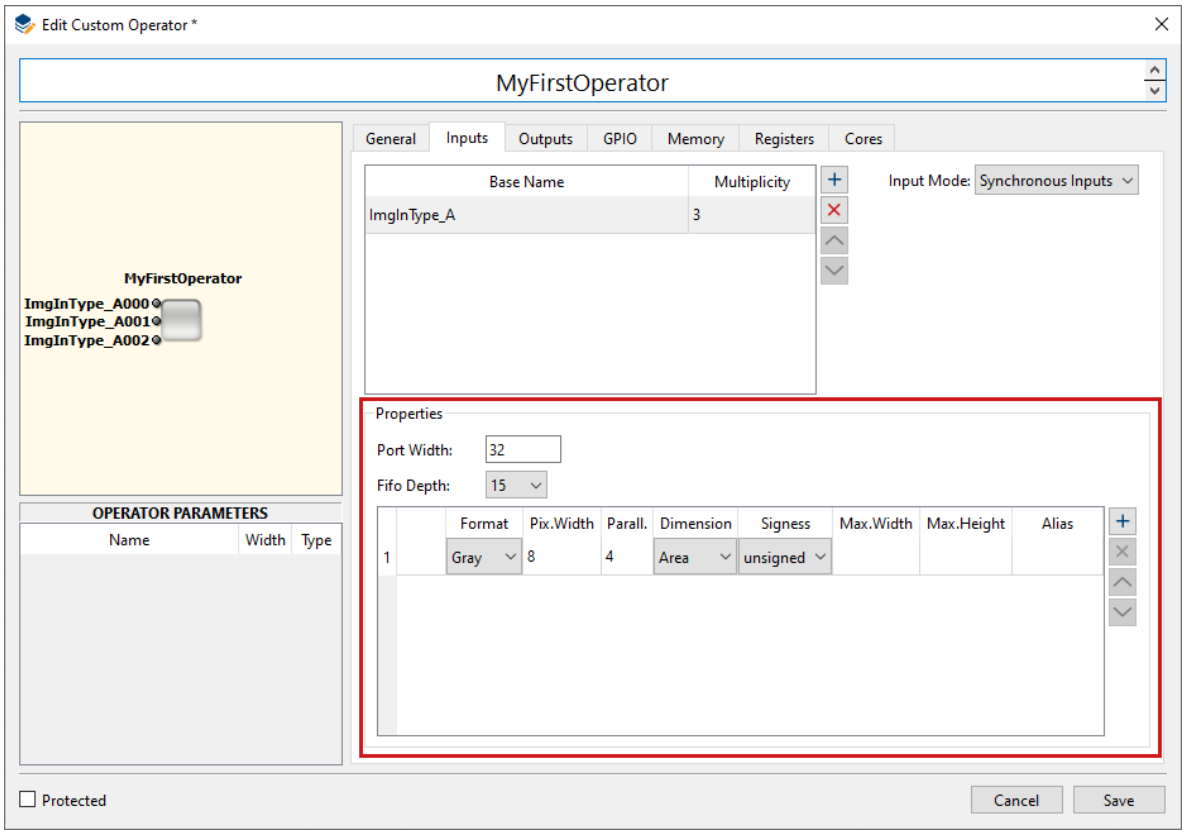
2. Give a name to the **ImgIn** port and define the number of input ports in the **Multiplicity** field: >1 defines an array of ports with a name consisting of the base name and an index.




Immediately, the operator depiction in the program window displays the entered array of **ImgIn** ports:




3. In the **Properties** panel, you specify the properties of the protocols that are supported by this **ImgIn** port.



4. Under **Port Width**, specify the width of the **ImgIn** port.

5. Under **Fifo Depth**, specify the depth of the buffer FIFO for input data which at least needs to be provided by the VA core. The value must be a power of two minus 1 between 15 and 1023.

   For an image interface port, you define a list of allowed protocols. A protocol can be described by the following properties:

   - Gray or color format (single or three data components with aggregated width **N**)
   - Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
   - Pixel data bit width N = [1..64]
   - Parallelism P = {1,2,4,8,16,32,64}
   - 2D (Aray), 1D (Line), or 0D (Raw)
   - Data type uint or int
   - Max. image dimensions

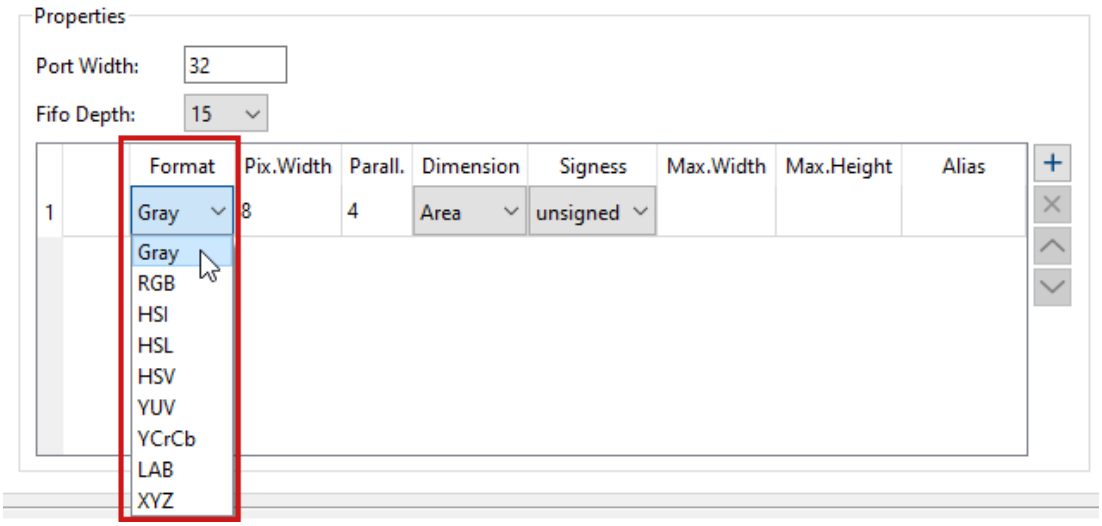
   Implicitly it is assumed that the kernel size is 1x1.

   The listed protocols are numbered starting from zero and therewith define an ID (in the image below visible in the left hand column of the table in the **Properties** panel).

   If you specify more than one protocol, you design the custom operator to support configuration of its input channel(s). In this case, several different protocols can be driven through a single port.

   The user of your custom operator can select the protocol he wants to use on a specific **ImgIn** port. According to the selection made by the VA user, the corresponding ID will be output to the related custom operator port. This enables the custom operator to adapt its behavior to the selected protocol.

6. Under **Format**, specify the color format of the protocol.




   The following color formats are allowed:

   - **grayXxP**: gray image with **X** bits per pixel and parallelism **P**
   - **rgbYxP:** color image with **Y**/3 bits per color component (red, green, blue) and parallelism **P**
   - **hsiYxP:** color image with **Y**/3 bits per color component (HSI color model) and parallelism **P**
   - **hslYxP:** color image with **Y**/3 bits per color component (HSL color model) and parallelism **P**

- **hsvYxP:** color image with **Y**/3 bits per color component (HSV color model) and parallelism **P**
- **yuvYxP:** color image with **Y**/3 bits per color component (YUV color model) and parallelism **P**
- **ycrcbYxP:** color image with **Y**/3 bits per color component (YCrCb color model) and parallelism **P**
- **labYxP:** color image with **Y**/3 bits per color component (LAB color model) and parallelism **P**
- **xyzYxP:** color image with **Y**/3 bits per color component (XYZ color model) and parallelism **P**

7. Double-click in the field of column **Pix.Width** and specify the pixel data width for the specific format:




The value range of *Pix.Width* depends on your choice under **Format**:

**Gray**: The pixel data width (in the following referred to as **X**) is limited to 64 bit.

**All color formats**: The pixel data width (in the following referred to as **Y**) must be a multiple of 3 and is limited to 63 bit.

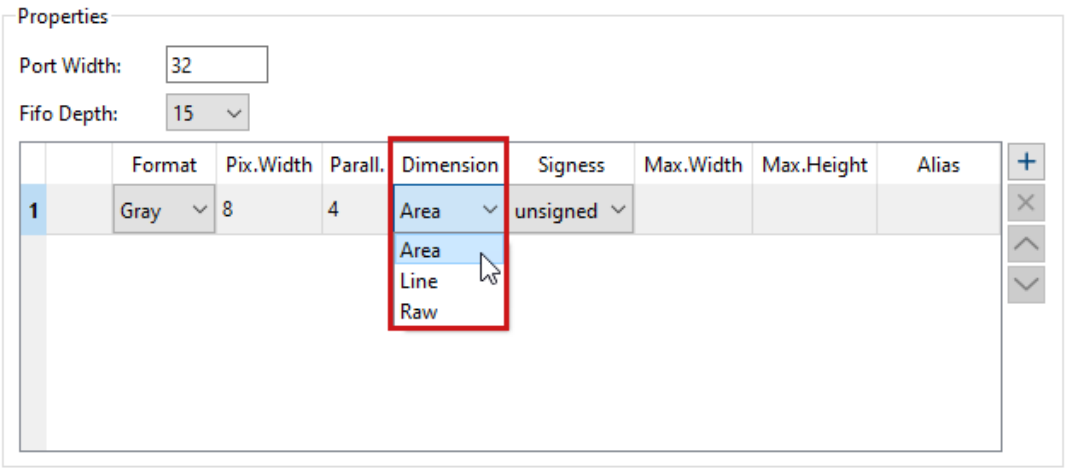
8. Double-click in the field of column **Parall.** and specify the parallelism for the specific format.

The parallelism defines the number of pixels which are contained in a single data word at the interface port. It must be chosen from following set of allowed values: **P** = {1, 2, 4, 8, 16, 32, 64}. Packing of image data into words of a given interface width **N** (specified under **Port Width**) must follow certain rules:

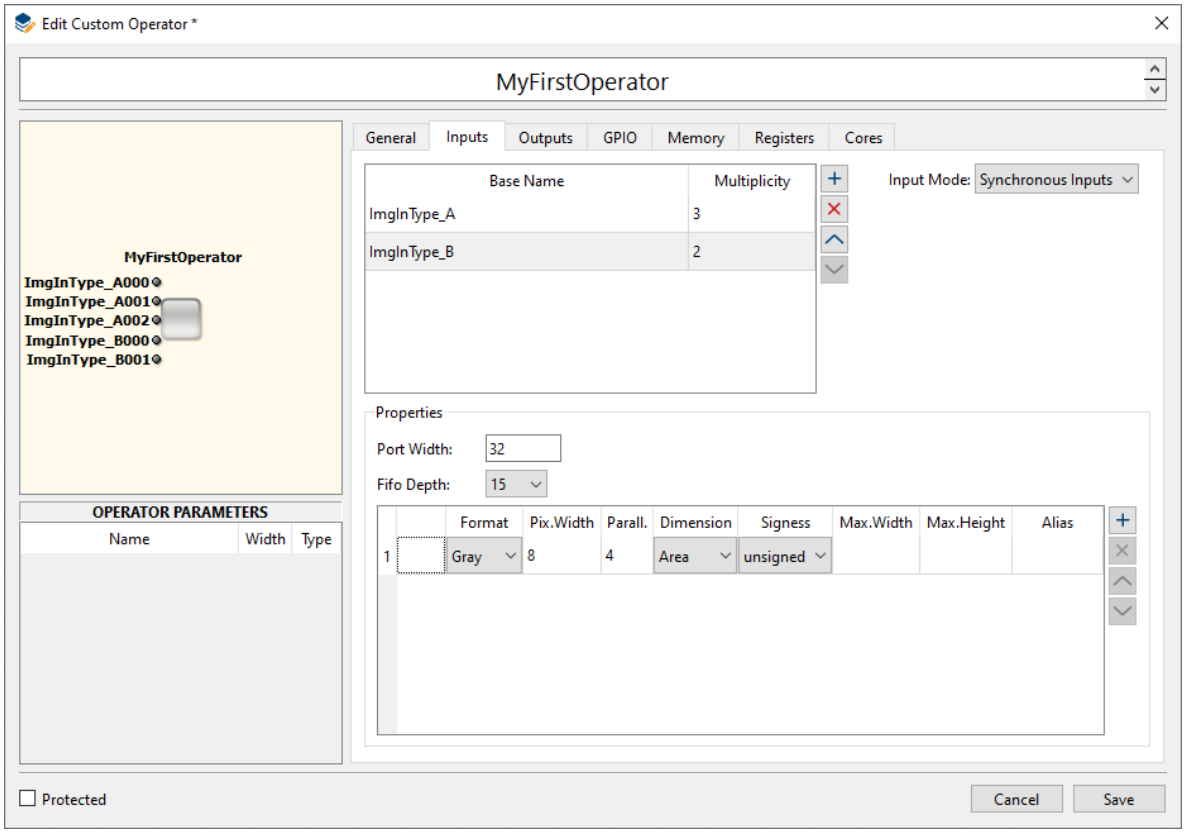
- The data of all **P** pixels must fit in a single word of length **N**. The data is stored LSB aligned which means that for a pixel width **Z** (**Z**=**X** for grey, **Z**=**Y** for color) data is distributed as follows: Pixel[0]->Bits[0..**Z**-1] .. Pixel[**P**-1]->Bits[(**P**-1)*Z..**P***Z-1].
- For RGB images the three color components are packed LSB aligned into a sub word [0..Y-1] in the following order: red uses the bits [0..**Y**/3-1 ], green the bits [**Y**/3..2*Y/3-1] and blue the bits [2*Y/3..3*Y/3-1].
- For HSI color images the same rules than for RGB applies where H takes the role of red, S that of green and I the role of blue.
- For HSL color images the same rules than for RGB applies where H takes the role of red, S that of green and L the role of blue.
- For HSV color images the same rules than for RGB applies where H takes the role of red, S that of green and V the role of blue.
- For YUV color images the same rules than for RGB applies where Y takes the role of red, U that of green and V the role of blue.

- For YCrCb color images the same rules than for RGB applies where Y takes the role of red, Cr that of green and Cb the role of blue.
- For LAB color images the same rules than for RGB applies where L takes the role of red, A that of green and B the role of blue.
- For XYZ color images the same rules than for RGB applies where X takes the role of red, Y that of green and Z the role of blue.

9. Under **Dimension**, specify if the protocol supports 2D (Area), 1D (Line), or 0D (Raw) images.




10. **Max.Width/Max.Height**: Using these optional fields you can define constraints for the image width and image height.
11. Repeat steps 6 to 10 to define as many protocols as you want the **ImgIn** port to support.
12. Repeat steps 1 to 11 to define as many **ImgIn** ports you want your custom operator to provide.

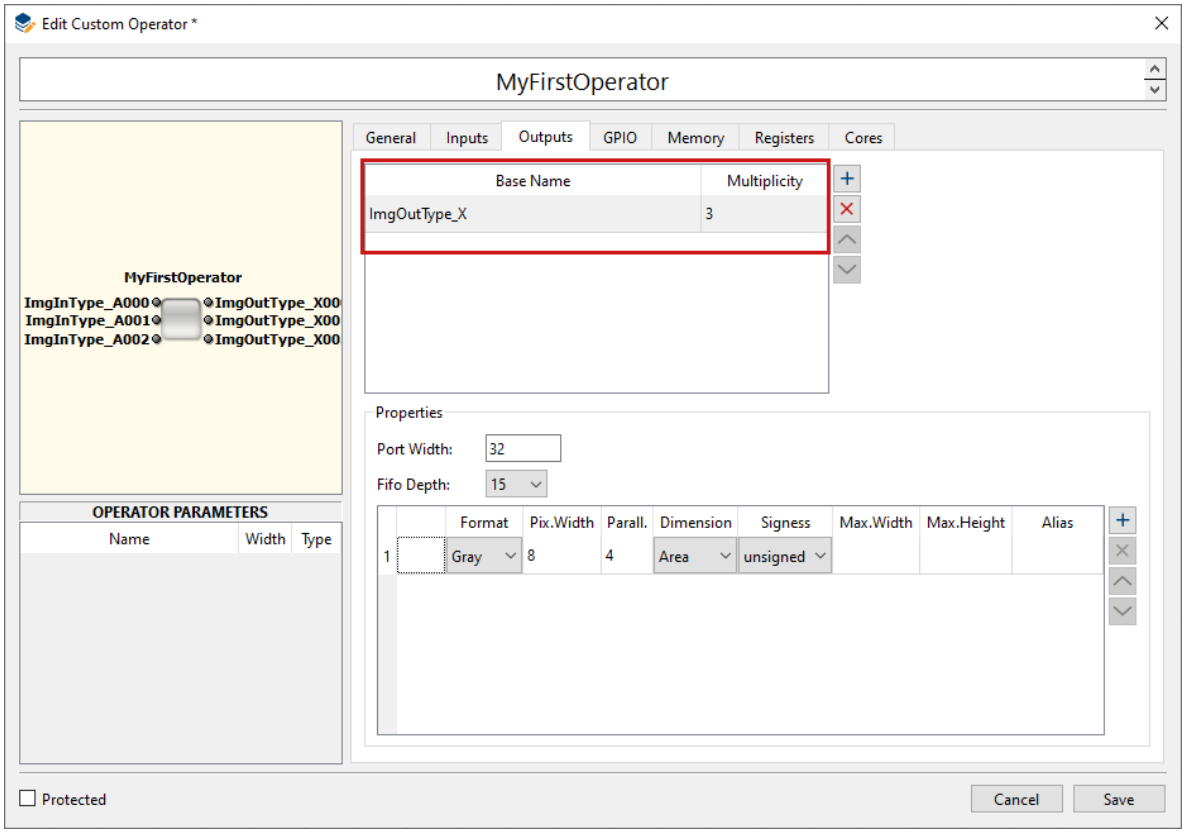



## 4.3.6.5. Defining the Image Output Ports

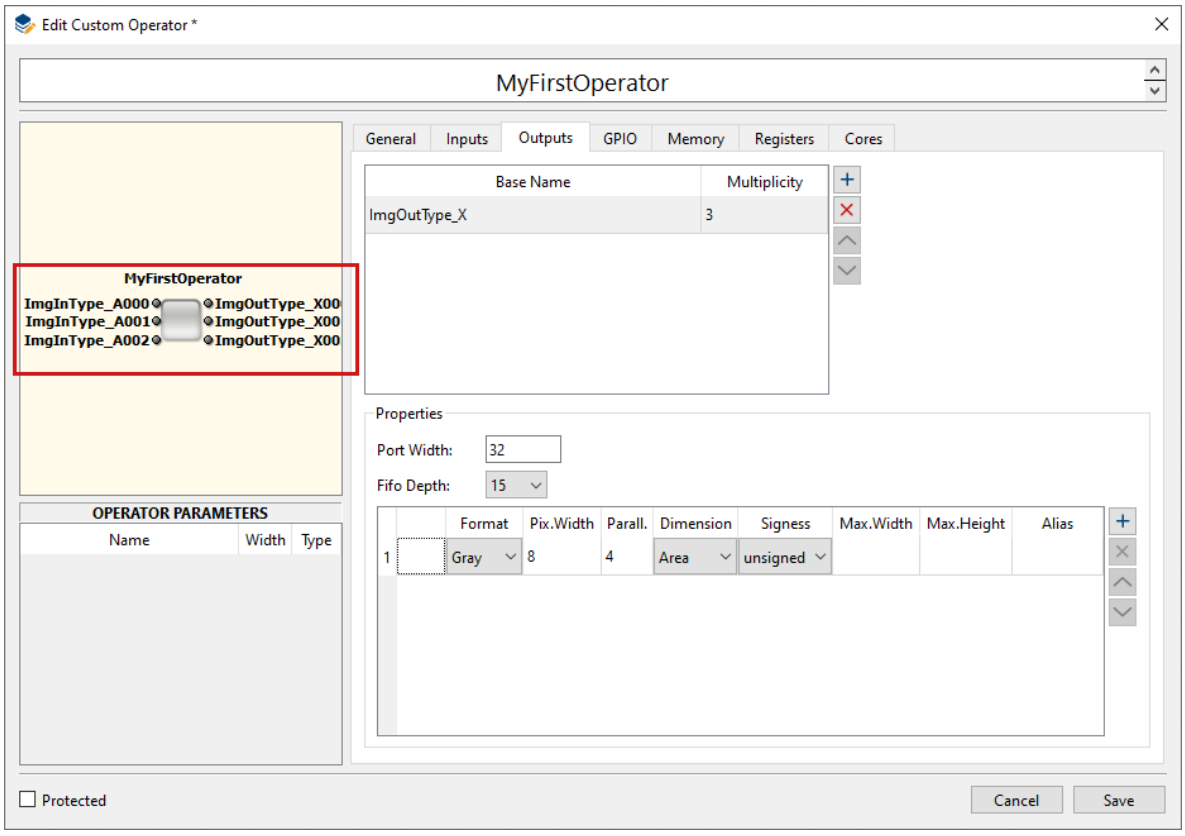
Under tab **Outputs**, you describe the properties of the image output ports.

You can define one or more image output ports (**ImgOut**). Each **ImgOut** port may be used as often as you specify.

1. Click on the **Plus** button + to create a first image out (**ImgOut**) port.
2. Give a name to the **ImgOut** port.
3. Double-click in the field of the **Multiplicity** column to create an array of ports. Multiplicity >1 defines an array of ports with a name consisting of the base name and an index.




Immediately, the operator depiction in the program window displays the entered array of **ImgOut** ports:



4. In the **Properties** panel, you specify the properties of the protocols that are supported by this **ImgOut** port.

   Under **Port Width**, specify the width of the **ImgOut** port.

5. Under **Fifo Depth**, specify the depth of the buffer FIFO for output data which at least needs to be provided by the VA core. The value must be a power of two minus 1 between 15 and 1023.

   For an image interface port, you define a list of allowed protocols. A protocol can be described by the following properties:

   - Gray or color format (single or three data components with aggregated width N)
   - Flavor of color format (RGB, HSI, HSL, HSV, YUV, YCrCb, LAB, XYZ)
   - Pixel data bit width N = [1..64]
   - Parallelism P = {1,2,4,8,16,32,64}
   - 2D (Aray), 1D (Line), or 0D (Raw)
   - Data type uint or int
   - Max. image dimensions

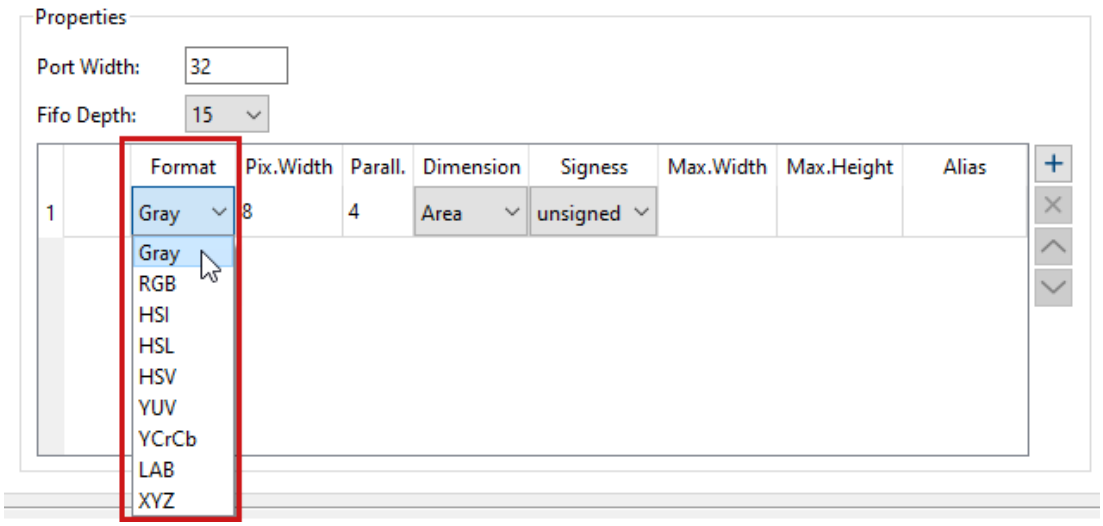
   Implicitly it is assumed that the kernel size is 1x1.

   The listed protocols are numbered starting from zero and therewith define an ID (in the image below visible in the left hand column of the table in the **Properties** panel).

   If you specify more than one protocol, you design the custom operator to support configuration of its input channel(s). In this case, several different protocols can be driven through a single port.

   The user of your custom operator can select the protocol he wants to use on a specific **ImgOut** port. According to the selection made by the VA user, the corresponding ID will be output to the related custom operator port. This enables the custom operator to adapt its behavior to the selected protocol.

6. Under **Format**, specify the color format of the protocol.

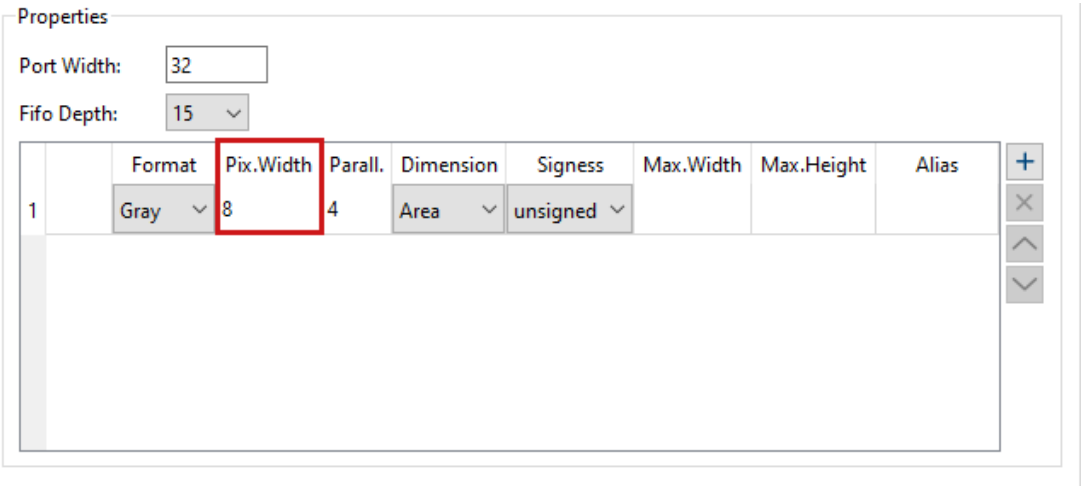



   The following color formats are allowed:

   - **grayXxP**: gray image with **X** bits per pixel and parallelism **P**
   - **rgbYxP:** color image with **Y**/3 bits per color component (red, green, blue) and parallelism **P**
   - **hsiYxP:** color image with **Y**/3 bits per color component (HSI color model) and parallelism **P**

- **hslYxP:** color image with **Y**/3 bits per color component (HSL color model) and parallelism **P**
- **hsvYxP:** color image with **Y**/3 bits per color component (HSV color model) and parallelism **P**
- **yuvYxP:** color image with **Y**/3 bits per color component (YUV color model) and parallelism **P**
- **ycrcbYxP:** color image with **Y**/3 bits per color component (YCrCb color model) and parallelism **P**
- **labYxP:** color image with **Y**/3 bits per color component (LAB color model) and parallelism **P**
- **xyzYxP:** color image with **Y**/3 bits per color component (XYZ color model) and parallelism **P**

7. Double-click in the field of column **Pix.Width** and specify the pixel data width for the specific format:




The value range of **Pix.Width** depends on your choice under **Format**:

**Gray**: The pixel data width (in the following referred to as **X**) is limited to 64 bit.

**All color formats**: The pixel data width (in the following referred to as **Y**) must be a multiple of 3 and is limited to 63 bit.

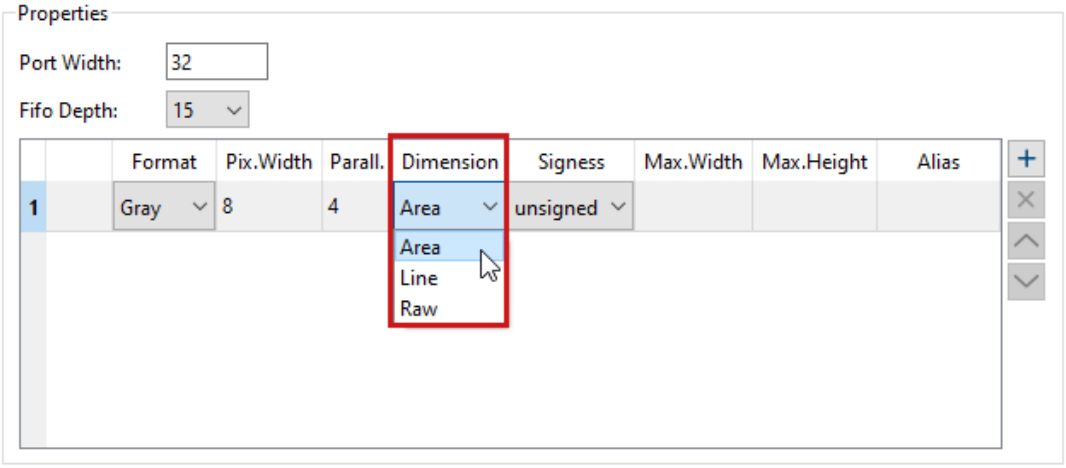
8. Double-click in the field of column **Parall.** and specify the parallelism for the specific format.

The parallelism defines the number of pixels which are contained in a single data word at the interface port. It must be chosen from following set of allowed values: **P** = {1, 2, 4, 8, 16, 32, 64}. Packing of image data into words of a given interface width **N** (specified under **Port Width**) must follow certain rules:

- The data of all **P** pixels must fit in a single word of length **N**. The data is stored LSB aligned which means that for a pixel width **Z** (**Z**=**X** for grey, **Z**=**Y** for color) data is distributed as follows: Pixel[0]->Bits[0..**Z**-1] .. Pixel[**P**-1]->Bits[(**P**-1)*__Z__..**P*****Z**-1].
- For RGB images the three color components are packed LSB aligned into a sub word [0..Y-1] in the following order: red uses the bits [0..**Y**/3-1 ], green the bits [**Y**/3..2***Y**/3-1] and blue the bits [2***Y**/3..3***Y**/3-1].
- For HSI color images the same rules than for RGB applies where H takes the role of red, S that of green and I the role of blue.
- For HSL color images the same rules than for RGB applies where H takes the role of red, S that of green and L the role of blue.
- For HSV color images the same rules than for RGB applies where H takes the role of red, S that of green and V the role of blue.

- For YUV color images the same rules than for RGB applies where Y takes the role of red, U that of green and V the role of blue.
- For YCrCb color images the same rules than for RGB applies where Y takes the role of red, Cr that of green and Cb the role of blue.
- For LAB color images the same rules than for RGB applies where L takes the role of red, A that of green and B the role of blue.
- For XYZ color images the same rules than for RGB applies where X takes the role of red, Y that of green and Z the role of blue.

9. Under **Dimension**, specify if the protocol supports 2D (Area), 1D (Line), or 0D (Raw) images.




10. **Max.Width/Max.Height**: Using these optional fields you can define constraints for the image width and image height.

11. Repeat steps 6 to 10 to define as many protocols as you want the **ImgOut** port to support.

12. Repeat steps 1 to 11 to define as many **ImgOut** ports you want your custom operator to support.

## 4.3.6.6. Defining the GPIO Ports

The General Purpose I/O interface allows connecting dedicated signal pins of the custom operator. Every GPIO port maps to a pin of the custom operator which is either an input or an output.

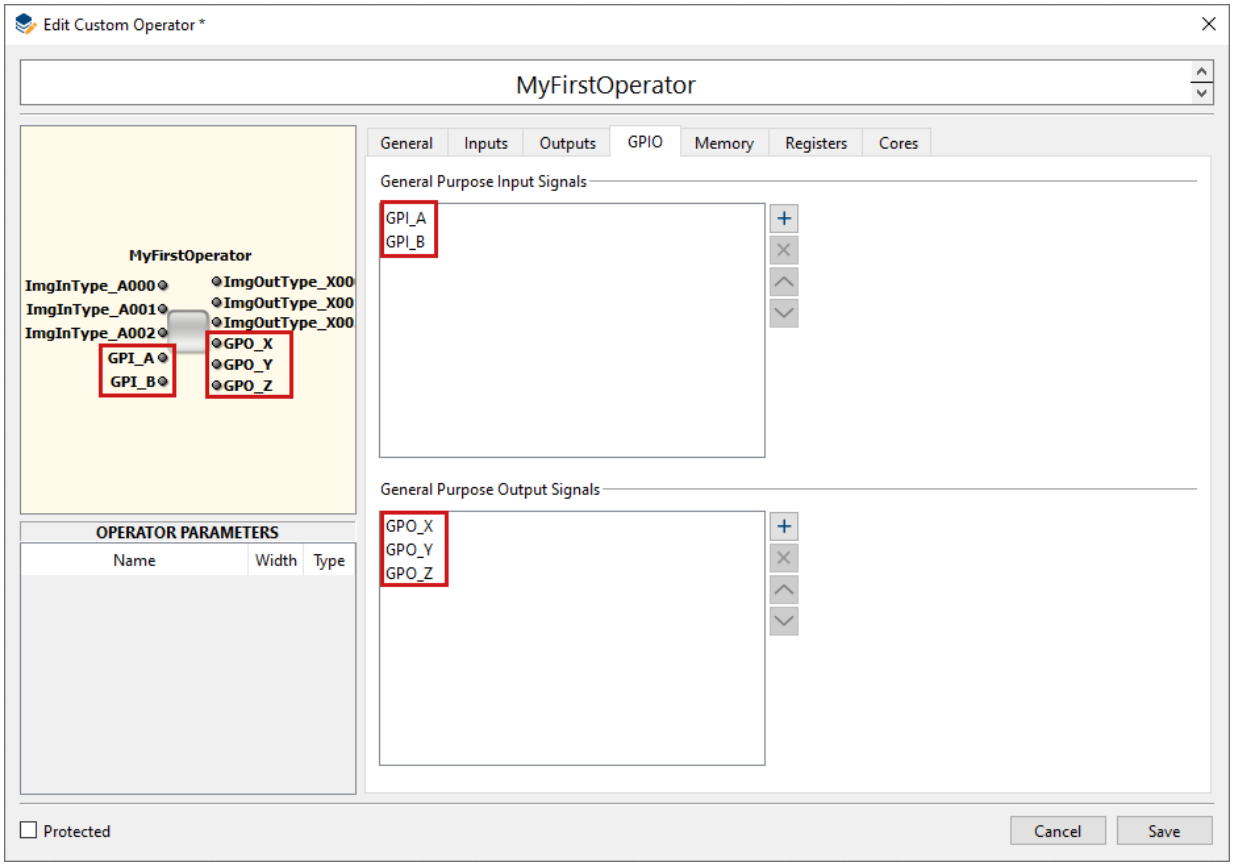
Bidirectional pins are not supported. In VisualApplets, the corresponding operator ports are of type *SIGNAL*.

1. Go to tab **GPIO**.

2. Add as many GPIs and GPOs as you want, using the **Plus** button +.

3. Double-click into the field to give a name to a specific GPI or GPO.

The defined GPIs and GPOs are immediately displayed in the depiction of the custom operator in the upper left hand panel of the program window:




**Bidirectional Pins Not Supported**

The pins are either an input or an output. Bidirectional pins are not supported.

## 4.3.6.7. Defining the Memory Ports

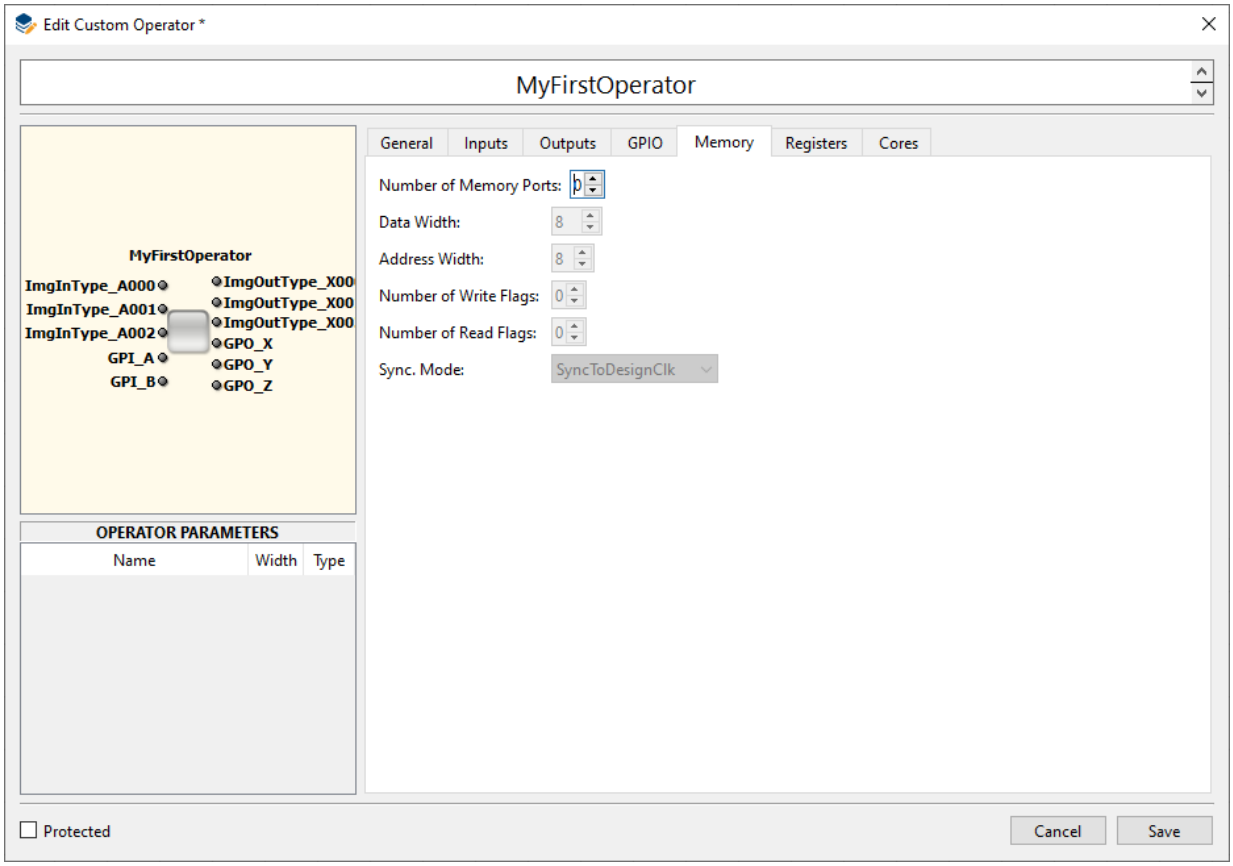
A custom operator may be set up for accessing one or more banks of memory (DRAM, SRAM, …).

All memory ports have a FIFO-like interface for write and read commands. The FIFOs reside in the VA part of the custom operator, so that you only need to implement a flow control, but not the FIFO. The timing of forwarding the FIFO content to the memory controller attached to the custom operator is fully controlled by VisualApplets.

Under the **Memory** tab, you can define that your operator gets access to external memory. You can specify up to 4 ports. You can specify the memory interface properties the operator needs.

**Comply with Memory Layout of Target Platforms**

Keep in mind the memory layout of potential target platforms (on which the applets containing the custom operator will run).




| Parameter Name | Type | Description |
|---|---|---|
| `Data Width` | Integer | Data width |
| `Address Width` | Integer | Address width |
| `Number of Write Flags (Width)` | Integer | Width of flag for marking write accesses. This parameter must be >= 1. |
| `Number of Read Flags (Width)` | Integer | Width of flag for marking read accesses. This parameter must be >= 8. |
| `SyncMode` | String | This parameter signals the relation of the memory interface clock and the design clock. Following values are possible:<br><br>`SyncToDesignClk` – memory interface ports are synchronous to `iDesignClk`.<br><br>`SyncToDesignClk2x` – memory interface ports are synchronous to `iDesignClk2x`. |

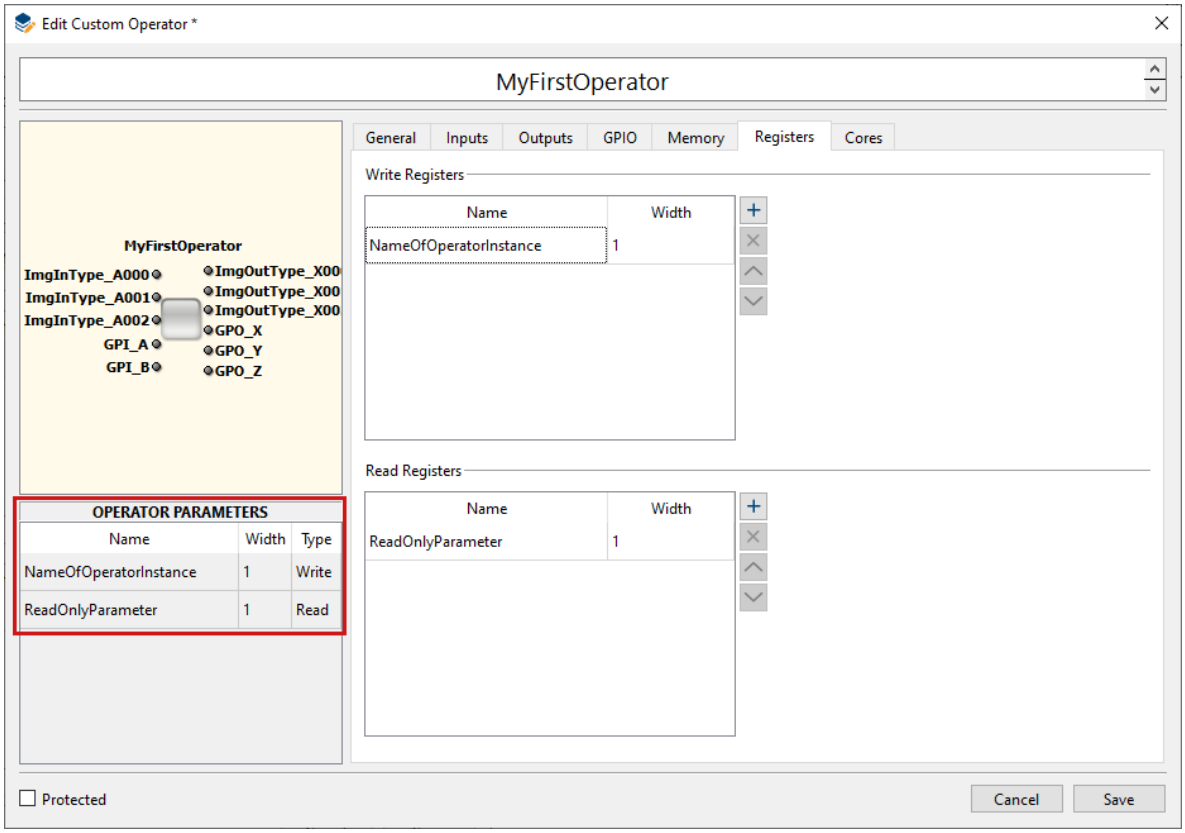
## 4.3.6.8. Defining the Registers of the Custom Operator

Under the **Registers** tab, you can define the write and read registers your custom operator will provide. Each of this registers is accessed in VisualApplets via a dedicated operator parameter. (The parameter name is the same as the register name.)

1. Go to the **Registers** tab.

2. Under **Write Registers**, define the write registers you want your custom operator to have.
3. Define a specific width for each write register.
4. Under **Read Registers**, define the read register you want your custom operator to have.
5. Define a specific width for each read register.

   The related operator parameters are immediately displayed in the left hand lower panel of the dialog window:



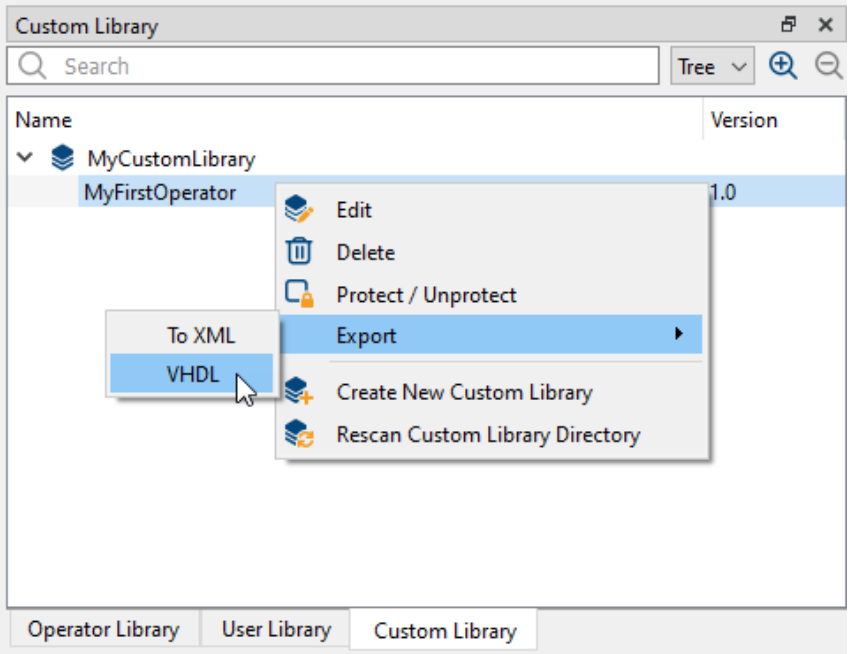

6. Click **Save.**

## 4.3.7. Generation of VHDL Black Box and Test Bench

After you have entered all details as described in section Section 4.3.6, 'Defining an Individual Custom Operator via GUI', you are ready for the actual VHDL coding. First of all, you need VisualApplets to generate the VHDL black box and test bench.

To trigger VHDL black box and test bench generation:

1. In the **Library** panel of the VisualApplets program window, go to the **Custom Library** tab.
2. Open the custom library and select the custom operator you want to implement.
3. Right-click on the operator name, and from the sub menu, select **Export** -> **VHDL**.




4. Specify the folder where you want the created VHDL files to be stored.

Now, the generation starts. After successful generation, a confirmation dialog opens up. Click **OK** to confirm.

You find all generated files in the folder you specified.

### 4.3.8. Operator Interface Ports

The generated black box provides all ports you specified via the GUI (see Section 4.3.6, 'Defining an Individual Custom Operator via GUI').

In this chapter, you find a detailed description of how these ports look like in the generated VHDL black box.

#### 4.3.8.1. Clock System, Reset and Enable

VisualApplets supports two clock domains. There is a base design clock and one derived clock which is in phase with that clock and has double frequency. Accordingly, there are two clock inputs to the custom operator. Additionally, there is a `Reset` and `Enable` input as described above.

| Port | Direction | Width | Description |
|---|---|---|---|
| `iDesignClk` | In | 1 | Base design clock |
| `iDesignClk2x` | In | 1 | Clock sync. to `iDesignClk` but double frequency |
| `iReset` | In | 1 | Reset of operator |
| `iEnable` | In | 1 | Enable processing |

#### 4.3.8.2. Parameter Interface

The definition of write register ports as described in section Section 4.3.6, 'Defining an Individual Custom Operator via GUI' leads to an interface as follows where **PORTID** is the register name and **PORTID**`Width` is the defined register width.

| Port | Direction | Width | Description |
|---|---|---|---|
| ivReg_**PORTID**_D | In | **PORTID**Width | Register data |
| iReg_**PORTID**_Wr | In | 1 | Signal write access |

The definition of read register ports as described in Section 4.3.6.8, 'Defining the Registers of the Custom Operator' leads to the following interface, accordingly:

| Port | Direction | Width | Description |
|---|---|---|---|
| ovReg_**PORTID**_D | Out | **PORTID**Width | Register data |
| iReg_**PORTID**_Rd | In | 1 | Signal read access |

### 4.3.8.3. Image Communication Interfaces

For communication of data between the VisualApplets core and a custom operator, image communication ports as described in Section 4.3.6.4, 'Defining the Image Input Ports' may be configured. Communication is done via a simple FIFO interface and an additional format identifier port.

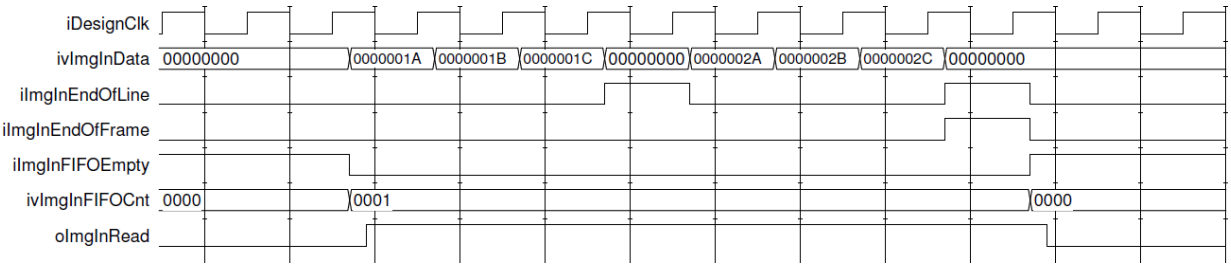
#### 4.3.8.3.1. Interfaces of Type ImgIn

An **ImgIn** channel for transferring data from the VisualApplets core to a custom operator leads to an interface as follows where **PORTID** is the name of the corresponding port type name and **X** is a port number for differentiating several ports of the same kind:

| Port | Direction | Width | Description |
|---|---|---|---|
| iv**PORTIDX**Data | In | **PORTID**Width | Data entering the custom operator |
| o**PORTIDX**Read | Out | 1 | Accept input data |
| i**PORTIDX**EndOfLine | In | 1 | Signal end of line. If this flag is activated data doesn’t contain pixel values. |
| i**PORTIDX**EndOfFrame | In | 1 | Signal end of frame. If this flag is activated data doesn’t contain pixel values. This flag is only asserted when end of line is signaled as well. |
| i**PORTIDX**FIFOEmpty | In | 1 | Buffer FIFO is empty |
| iv**PORTIDX**FIFOCnt | In | Ceil Log2(**PORTID**FIFODepth) | Number of words in buffer FIFO. This signal can be used to generate FIFO flags like Almost Empty. |
| iv**PORTIDX**_FID_D | In | Ceil Log2(N) | Predefined parameter which notifies about the current image data format. **N** is the number of image formats specified for this port. |

The figure below illustrates the data flow at an **ImgIn** port. The port name component **PORTIDX** has been substituted by **ImgIn**. The waveform shows the input of a two dimensional frame of size 3x2.

When the **ImgIn** port is part of several O-synchronous input ports, all of them must consume the FIFO data simultaneously. In that case the FIFO fill level of all ports will exactly match so the operator only needs to implement flow control according to the fill level of one out of several O- synchronous inputs.

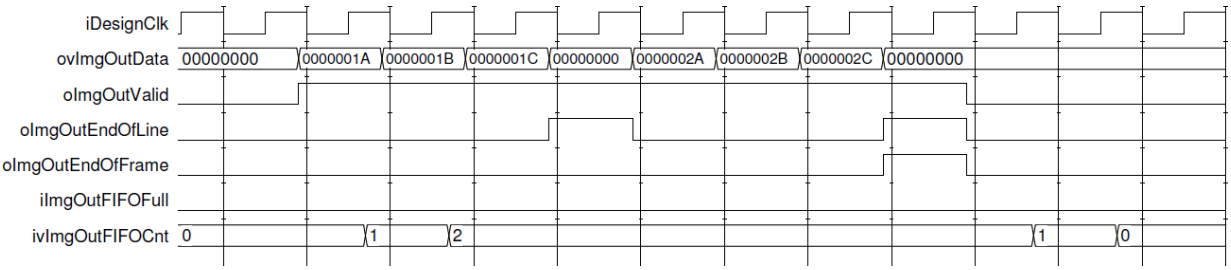



### 4.3.8.3.2. Interfaces of Type ImgOut

An **ImgOut** channel for transferring data from a custom operator to the VisualApplets core leads to an interface as follows where **PORTID** is the name of the corresponding port type name and **X** is a port number for differentiating several ports of the same kind:

| Port | Direction | Width | Description |
|---|---|---|---|
| ov**PORTIDX**Data | Out | **PORTID**Width | Output data |
| o**PORTIDX**Valid | Out | 1 | Output data valid |
| o**PORTIDX**EndOfLine | Out | 1 | Signal current write access as end of line notification. Write data is then not interpreted as pixel data. |
| o**PORTIDX**EndOfFrame | Out | 1 | Signal current write access as end of frame notification. Write data is then not interpreted as pixel data. This flag needs to be correlated with an end of line strobe at the same time. |
| i**PORTIDX**FIFOFull | In | 1 | Buffer FIFO is full, no further data is accepted |
| iv**PORTIDX**FIFOCnt | In | Ceil Log2(**PORTID**FIFODepth) | Number of words in buffer FIFO. This signal can be used to generate FIFO flags like Almost Full. |
| iv**PORTIDX**_FID_D | In | Ceil Log2(N) | Predefined parameter which notifies about the current image data format. **N** is the number of image formats specified for this port. |

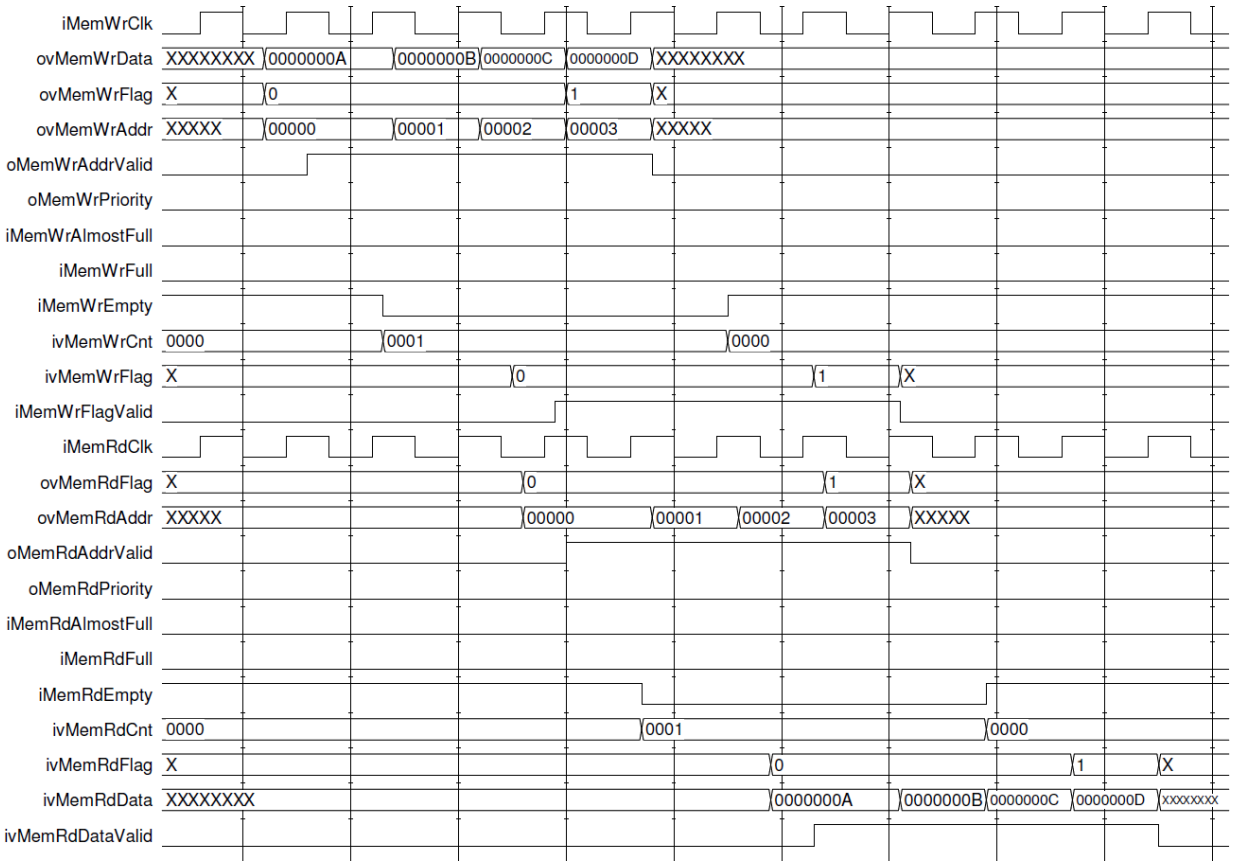
The figure below illustrates the data flow at an **ImgOut** port. The waveform shows the output of a two dimensional frame of size 3x2. When the **ImgOut** port is part of several O-synchronous output ports all of them must emit data simultaneously.



### 4.3.8.4. Memory Interfaces

A custom operator may be set up for having up to four memory ports. The I/O ports of the generated interface get a suffix **X** where **X** is the index of the memory port.

| Name | Direction | Width | Description |
|---|---|---|---|
| ovMemWrData**X** | Out | MemDataWidth**X** | Write data output to memory via VisualApplets core |
| ovMemWrFlag**X** | Out | MemWrFlagWidth**X** | Write flag output |
| ovMemWrAddr**X** | Out | MemAddrWidth**X** | Write address |
| oMemWrAddrValid**X** | Out | 1 | Emit write command |
| oMemWrPriority**X** | Out | 1 | Request priority for this write port |
| iMemWrAlmostFull**X** | In | 1 | Only single further write command may be accepted |
| iMemWrFull**X** | In | 1 | No write command is accepted as concerning FIFO is full |
| iMemWrEmpty**X** | In | 1 | FIFO for write commands is empty |
| ivMemWrCnt**X** | In | 4 | Number of buffered write commands |
| ivMemWrFlag**X** | In | MemWrFlagWidth**X** | Write flag output from the VisualApplets core |
| iMemWrFlagValid**X** | In | 1 | Write flag input valid – signals that iMemWrFlag**X** is valid, which means that write access which had been marked with corresponding oMemWrFlag**X** has been executed. |
| ovMemRdFlag**X** | Out | MemRdFlagWidth**X** | Read flag |
| ovMemRdAddr**X** | Out | MemAddrWidth**X** | Read address |
| oMemRdAddrValid**X** | Out | 1 | Emit read command |
| oMemRdPriority**X** | Out | 1 | Request priority for this read port |
| iMemRdAlmostFull**X** | In | 1 | Only single further read command may be accepted |
| iMemRdFull**X** | In | 1 | No read command is accepted as concerning FIFO is full |
| iMemRdEmpty**X** | In | 1 | FIFO for read commands is empty |
| ivMemRdCnt**X** | In | 4 | Number of buffered read commands |
| ivMemRdFlag**X** | In | MemRdFlagWidth**X** | Read flag input – only valid when iMemRdDataValid**X** is asserted |
| ivMemRdData**X** | In | MemDataWidth**X** | Read data input |
| iMemRdDataValid**X** | In | 1 | Read data valid |




The figure above illustrates the waveform of the memory interface protocol.

### 4.3.8.5. General Purpose I/O pins

Any GPIO input or output signal which has been defined in the interface description of the custom operator (section Section 4.3.6, 'Defining an Individual Custom Operator via GUI') has a corresponding input or output port in the resulting operator interface. The following ports will be created when the general purpose pins are declared:

- `iSig_`**NAME** for a GPIO input signal called **NAME**.
- `oSig_`**NAME** for a GPIO output signal called **NAME**.

## 4.3.9. VHDL Simulation and Verification

For emulating a VisualApplets design which contains a custom operator module, VisualApplets creates a simulation test bench for the interfaces connecting to the custom operator.

Each interface port is emulated independently, driven by File I/O. The simulation entity shall consist of following elements:

- Emulation of register access. According to a stimuli file a set of registers can be written and read.
- Emulator for frame source connected to ports of type **ImgIn**. Stimulated by file these kinds of modules output frame data to **ImgIn**.
- Emulator for frame sink connected to ports of type **ImgOut**. This kind of module emulates an operator which is connected to **ImgOut**. The module writes the received data to file.
- Memory port emulator.

- GPIO emulator. Each GPIO signal for input is driven by a signal generator which is configured by a file. Each GPIO signal output is monitored and changes of the signal are written to a report file.

### 4.3.9.1. Simulation Framework

For RTL level simulation, VisualApplets creates a VHDL file containing a package with the name `CustomOperator_`**<OPERATORNAME>** where **<OPERATORNAME>** is the given operator name.

This package contains the components **<OPERATORNAME>** and **<OPERATORNAME>**_TB where the latter is a test bench of the interface between the VisualApplets design and the custom operator. The following shows the resulting code for a simple custom operator called *RegExample* consisting only of a read and write register port (`Ctrl` and `Status`), each 4 bit wide:

```
component
RegExample port(
  iDesignClk: in std_logic := '0';
  iDesignClk2x: in std_logic := '0';
  iReset: in std_logic := '0';
  iEnable: in std_logic := '0';
  ivReg_Ctrl_D: in std_logic_vector(3 downto 0);
  iReg_Ctrl_Wr: in std_logic;
  ovReg_Status_D: out std_logic_vector(3 downto 0);
  iReg_Status_Rd: in std_logic
);
end component;

component
RegExample_TB generic(
  DesignClkPeriod: time := 16 ns;
  Register_StimuliFileName: string := ""
);
end component;
```

The test bench creates an instance of the custom operator and connects protocol emulation modules to each interface ports. The following sections describe the different kinds of emulators, how they may be controlled via stimuli files, and how output files are generated.

### 4.3.9.2. Emulation of Register Interface

The generated test bench implements an emulator for a register access interface. The emulator is configured for addressing a design with a single process. Addresses of write and read registers start from 0x4 where addresses for registers are counted up with an increment of 1 according to the sequence of the register interface ports in the given custom operator component (like the above example component **RegExample**). Register addresses for reading and writing are counted independently. The emulator is driven by a text file which is set by the entity parameter `Register_StimuliFileName` as provided in the above VHDL code.

The following commands may be present in the stimuli file:

| Command | Description |
|---|---|
| `REM` | Rest of line is comment |
| `GRS` | Emulate global reset |
| `PRS` | Emulate process reset. This command has the following syntax: `PRS <procNr>` where the parameter `<procNr>` must always be 0. |
| `PEN` | Enable process. The syntax is as follows: `PEN <procNr> <value>` with `<procNr>` being always 0 and `<value>` signaling the enable state. |
| `WCK` | Wait for a number of clock cycles. The syntax is as follows, `WCK <clock_ticks>` with `<clock_ticks>` giving the number of clock ticks in hexadecimal format |
| `WRR` | Write to register: `WRR <wrRegAddr> <value>` With the parameters: `<wrRegAddr>`: address of register (hex) `<value>`: hexadecimal register value |

| Command | Description |
|---|---|
| RDR | Read from register: RDR <rdRegAddr> with <wrRegAddr> being the register address (hex). |

After the last parameter of any command, a comment may be added preceded by "#".

The following code is an example stimuli file which accesses the registers according to the above given test bench RegExample_TB:

```
REM ************************************************************
REM Command formats:
REM GRS -> Global reset
REM GEN <value> -> Set global enable to <value>
REM PRS <procID> -> Reset process <procID> (0 .. F) REM PEN <procID> <value>
  -> Set enable of process <procID> to
<value>
REM WCK <clk_ticks> -> Wait for <clk_ticks> clock cycles REM WRR <wrRegAddr> <value>
  -> Write <value> to register <wrRegAddr> REM RDR <rdRegAddr> -> Read from register <rdRegAddr>
REM ************************************************************

WCK 0004  # wait for 4 clock cycles
GRS   # global reset
GEN 1  # set global enable
WCK 0001  # wait for 1 clock tick
PRS 0  # reset process 0
PEN 0 1  # set enable of process 0
WCK 0002  # wait for 2 clock ticks
WRR 0004 0000000A # write 0xA to address 0x4
WCK 0002  # wait for 2 clock ticks
RDR 0004  # read from address 0x4
WCK FFFF
```

### 4.3.9.3. Emulation of ImgIn Interface

The emulation of image communication interfaces of type **ImgIn** is driven by a stimuli file providing information about the sequence of data which enters the custom operator. For any present **ImgIn** port the test bench has a generic **<PORTIDX>**_StimuliFileName where <**PORTIDX**> is the name of the corresponding image input port type followed by the port number. Each line within the given file must follow the syntax:

```
<Command> <Data> <EndOfLine> <EndOfFrame> <DataValid>
```

where <Command> is a three letter command, <Data> provides an hexadecimal data word, and the three remaining parameters correspond to the image protocol flags.

The following table describes the available commands:

| Command | Description |
|---|---|
| DAT | Data command. This command provides data which will become input at the port iv**PORTIDX**Data and the associated image protocol flag ports. |
| WCK | Wait command. The parameter <Data> provides the number of clock ticks for which the command interpreter pauses. |
| FID | Set FID input. The parameter <Data> provides the value to which the port iv**PORTIDX**_FID_D will be set. |

To any command line a comment may be added, preceded by "#".

The following code is an example stimuli file which causes the input of an 3x2-image:

```
FID 00000001 0 0 0 #Format: Cmd Data(hex) EndOfLine EndOfFrame DataValid
DAT 00000000 0 0 0
DAT 0000001a 0 0 1
DAT 0000001b 0 0 1
```

```
DAT 0000001c 0 0 1
DAT 00000000 1 0 1
WCK 00000004 0 0 0
DAT 0000002a 0 0 1
DAT 0000002b 0 0 1

DAT 0000002c 0 0 1
DAT 00000000 1 1 1
WCK 0000FFFF 0 0 0
```

### 4.3.9.4. Emulation of ImgOut Interface

The emulation of image communication interfaces of type **ImgOut** is driven by a stimuli file where information is provided about the sequence of FID states. For any present **ImgOut** port the VA_Design_Emulator entity has a generic **<PORTIDX>**_StimuliFileName where **<PORTIDX>** is the name of the corresponding image output port type followed by the port number. The syntax is exactly the same as in the case of the stimuli for **ImgIn** interfaces except that no DAT command is available. A simple stimuli file may look like:

```
WCK 00000010 0 0 0    #Format: Command Data(hex)
FID 00000001 0 0 0
WCK 0000FFFF 0 0 0
```

The parameters <EndOfLine>,<EndOfFrame> and <DataValid> are actually meaningless.

The **ImgOut** interface emulator present in the generated test bench writes the received data to file. For that purpose the test bench entity has a generic **<PORTIDX>**_DumpFileName. During simulation a file with the given name is created and the data is written using DAT and WCK commands in a format, which exactly corresponds to the stimuli file format for an **ImgIn** interface emulator.

### 4.3.9.5. Emulation of Memory Communication

When the custom operator implements an interface to memory the test bench connects a memory emulation module to the corresponding interface ports. The custom operator may not rely on a certain timing of the memory interface (like time until read data is returned) as this is fully controlled by VisualApplets and may vary between platforms and even between different designs.

### 4.3.9.6. GPIO Emulation

The emulation of dedicated input signals is done for each signal independently, driven by a stimuli file. There information is provided about the sequence of signal states. The stimuli file may consist of a number of commands which are described below. For any present output signal port the test bench entity has a generic iSig_**<NAME>**_StimuliFileName where **<NAME>** is the concerning port name.

The following table describes the available commands:

| Command | Description |
|---|---|
| SET | Set signal. This command provides the signal state to which the output at the port iSig_**NAME** will be set. The next command will be executed one clock tick later. It has the syntax: SET <value> where <value> may be 0 or 1. |
| WCK | Wait command. It has the syntax: WCK <ticks> where the parameter <ticks> provides the number of clock ticks for which the signal will be held constant. |
| RST | Restart from begin. The command interpreter will start again from the first line of the stimuli file. This command does not have any parameters. The command will execute the first command of the file at the same clock tick allowing assembling a loop without a gap. |
| STP | Stop at current state. The command interpreter will stop and the current signal state will be held constant until end of simulation. This command does not have any parameters. |

To any command line a comment may be added, preceded by "#".

The following code is an example stimuli file which causes the custom operator input signal toggling being low for 5 clock cycles and high for 7 clock cycles (synchronous to `iDesignClk`):

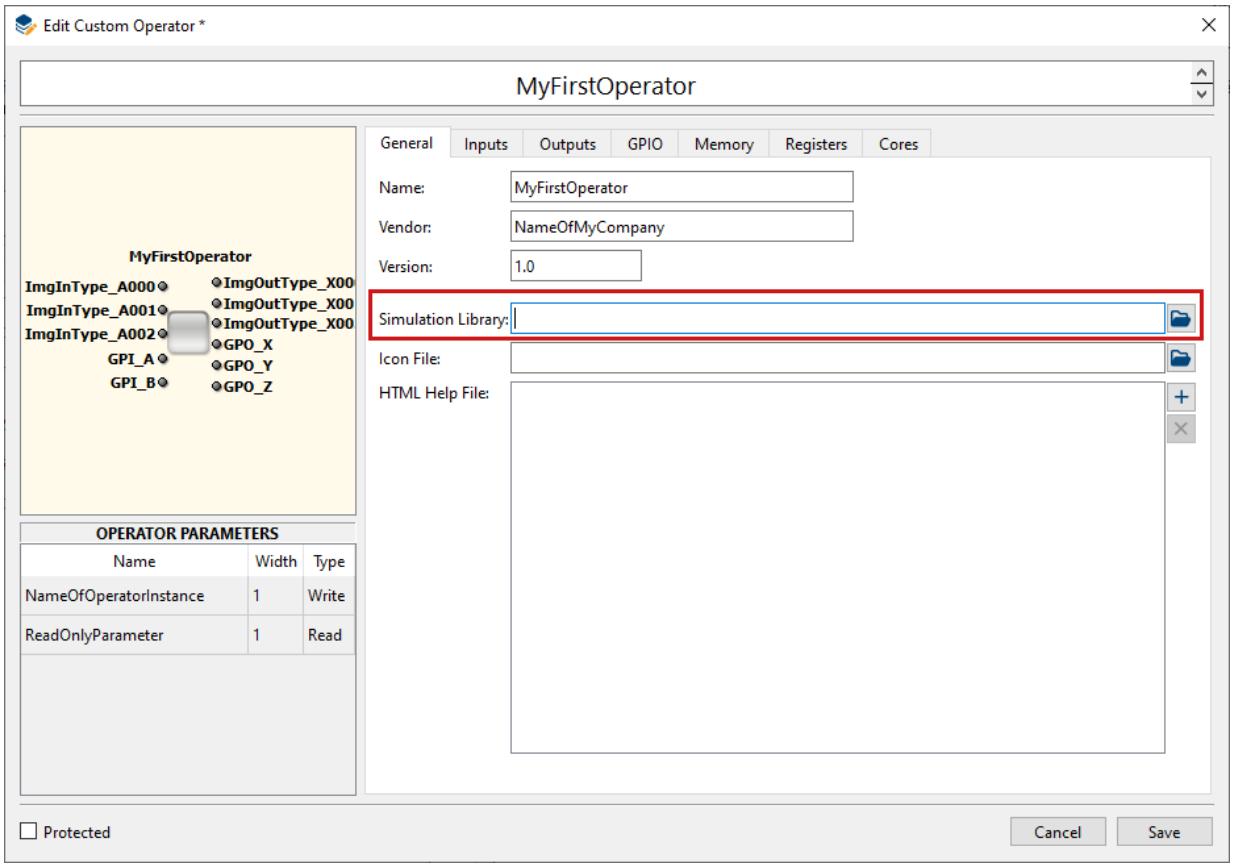
```
SET 0    # deassert output
WCK 0004 # wait for 4 clock cycles
SET 1    # assert output
WCK 0006 # wait for 6 clock cycles
RST      # restart from begin
```

Dedicated output signals are monitored writing a dump file `oSig_`**<NAME>**`_DumpFileName` where

**<NAME>** is the concerning port name. The file is composed of SET and WCK commands exactly corresponding to the commands of the stimuli file for an dedicated input signal.

## 4.3.10. Defining the Custom Operator's Software Interface

The high-level simulation component must be provided for fully integrating a custom operator to VisualApplets. The high-level simulation component needs to be compiled to a dynamic link library with a predefined set of exported C-Functions. You add this file to the operator specification under tab **General** -> **Simulation Library**:




For High-level simulation within VisualApplets the following function must be exported,

```
int SimulateOPNAME (va_custom_op_sim_handle simHandle)
```

where **OPNAME** is the name of the custom operator.

High-level simulation must be done according to following requirements:

- **Frame based simulation** - On each image input port it can be queried whether one or more frames are available. If all ports which are required for starting simulation are able to provide a frame then the concerning output frames need to be computed and emitted via calls of appropriate functions. For one dimensional image data the data stream is automatically split into frames and simulated just like 2D-data.
- **Bit accurate simulation** – The calculation of resulting frames must be bit accurate, i.e. the output data must be exactly equal to the data generated by the hardware implementation.
- **Keeping consistency of flow** – When operator input ports are synchronous to each other input images must be fetched accordingly. When several outputs are defined images must be output simultaneously. For the simulation function this means that when a frame is output to one output link it must also output a frame to all other output links before the simulation function is returning.

As the behavior of the operator typically depends on the set of operator parameters these parameters may be queried via the following interface:

| Nr. | Function | Description |
|---|---|---|
| 1 | `vaSi_CustomOp_GetParamValue()` | Get value of operator parameter. |

Several functions are provided by VisualApplets for getting, generating and storing image data for the custom operator:

| Nr. | Function | Description |
|---|---|---|
| 1 | `vaSi_CustomOp_GetInputImage()` | Get image available at an **ImgIn** port. |
| 2 | `vaSi_CustomOp_PutOutputImage()` | Output image to **ImgOut** port. |
| 3 | `vaSi_CustomOp_InputHasImage()` | Query whether **ImgIn** port may deliver an image. |
| 4 | `vaSi_CustomOp_OutputReady()` | Query whether **ImgOut** port may take an image. |
| 5 | `vaSi_CustomOp_CreateImage()` | Create new image. |
| 6 | `vaSi_CustomOp_DeleteImage()` | Delete image. |
| 7 | `vaSi_CustomOp_StoreImage()` | Store image in local storage of operator instance providing a name whereby the image may later be referenced. |
| 8 | `vaSi_CustomOp_GetStoredImagesCount()` | Query number of images stored within operator instance. |
| 9 | `vaSi_CustomOp_GetStoredImage()` | Get stored image by index. |
| 10 | `vaSi_CustomOp_GetNameOfStoredImage()` | Get name of stored image by index. |
| 11 | `vaSi_CustomOp_GetStoredImageByName()` | Get stored image by name. |
| 12 | `vaSi_CustomOp_CreateImageFormat()` | Create new image format handle which becomes initialized by the format associated with the given port. |
| 13 | `vaSi_CustomOp_CopyImageFormat()` | Create new image format which is a copy of given format. |
| 14 | `vaSi_CustomOp_DeleteImageFormat()` | Delete image format handle created earlier. |

For manipulating images via image handles the following functions are available:

| Nr. | Function | Description |
|---|---|---|
| 1 | `vaSi_Image_GetFormat()` | Get image format. |
| 2 | `vaSi_Image_SetProperty()` | Set property of frame (e.g. height). |
| 3 | `vaSi_Image_GetProperty()` | Get property of frame. |
| 4 | `vaSi_Image_SetPixelValue()` | Set pixel component value |
| 5 | `vaSi_Image_GetPixelValue()` | Get pixel component value |

| Nr. | Function | Description |
|---|---|---|
| 6 | vaSi_Image_SetLineLength () | Set individual length of a line. |
| 7 | vaSi_Image_GetLineLength () | Get length of individual line. |

Image formats may be manipulated via the following functions:

| Nr. | Function | Description |
|---|---|---|
| 1 | vaSi_ImageFormat_SetProperty() | Set image format property (e.g. maximum width). |
| 2 | vaSi_ImageFormat_GetProperty() | Get image format property. |

The simulation function may inject an status message (i.e., error message) into the VisualApplets simulation system using the following functions:

| Nr. | Function | Description |
|---|---|---|
| 1 | vaSi_CreateStatusMessage() | Create status message. |
| 2 | vaSi_SetStatusMessageProperty() | Set property of status message (like severity). |
| 3 | vaSi_SendStatusMessage() | Submit the status message to the simulation engine. |

### 4.3.11. Communicating Data

For querying information and configuring parameters data must be exchanged through the software interface. To keep the interface functions simple but providing a type save interface an abstraction mechanism for data is implemented. Whenever data of different types needs to be communicated, a data structure called va_data is used, containing a reference to the data and information about the underlying data type. This data structure is created by the user but configured by dedicated functions listed below. The following table shows the data types which are handled by this method:

| Data Type | Description |
|---|---|
| VA_ENUM | enum entry given as 32-Bit integer |
| VA_INT32 | 32-Bit signed integer |
| VA_UINT32 | 32-Bit unsigned integer |
| VA_INT64 | 64-Bit signed integer |
| VA_UINT64 | 64-Bit unsigned integer |
| VA_DOUBLE | Floating-point number, double precision |
| VA_INT32_ARRAY | Array of 32-Bit signed integer numbers |
| VA_UINT32_ARRAY | Array of 32-Bit unsigned integer numbers |
| VA_INT64_ARRAY | Array of 64-Bit signed integer numbers |
| VA_UINT64_ARRAY | Array of 64-Bit unsigned integer numbers |
| VA_DOUBLE_ARRAY | Array of double numbers |
| VA_STRING | String given as const char* |

Configuring an earlier created va_data structure (vaData) for setting up data communication is done via the following functions:

```
va_data* va_data_enum(va_data* vaData, int32_t *data)
va_data* va_data_int32(va_data* vaData, int32_t *data)
va_data* va_data_uint32(va_data* vaData, uint32_t *data)
va_data* va_data_int64(va_data* vaData, int64_t *data)
va_data* va_data_uint64(va_data* vaData, uint64_t *data)
va_data* va_data_double(va_data* vaData, double *data)
va_data* va_data_int32_array(va_data* vaData, int32_t *data, size_t elementCount)
va_data* va_data_uint32_array (va_data* vaData, uint32_t *data, size_t elementCount)
va_data* va_data_int64_array (va_data* vaData, int64_t *data, size_t elementCount)
```

```
va_data* va_data_uint64_array (va_data* vaData, uint64_t *data, size_t elementCount)
va_data* va_data_double_array (va_data* vaData, double *data, size_t elementCount)
va_data* va_data_string(va_data* vaData, char data*, size_t strSize)
va_data* va_data_const_string(va_data* vaData, const char **data)
```

For strings there are two options how strings are communicated:

1. Providing a char array via `va_data_string()`. Then queried string data will be copied to that array.
2. Providing a pointer to `const char*`. Then a pointer to an internal string representation is returned when information of type `VA_STRING` is queried. When you use this approach check the livetime of the returned string.

Example Code: The following example shows code for querying the image width.

```
uint32_t imgWidth;
va_data
va_imgWidth;
va_data_double(&va_imgWidth,&imgWidth);

vaSi_Image_GetProperty(imageHandle, "Width", &va_imgWidth);
```

After that the variable `imgWidth` will contain the requested information.

### 4.3.12. Detailed Description of Interface Functions

The following gives a detailed description of parameters and returned values for the specified simulation interface functions.

| **Function** | **vaSi_CustomOp_GetParamValue** |
|---|---|
| Syntax | `int vaSi_CustomOp_GetParamValue (va_custom_op_sim_handle simHandle, const char* paramName, va_data *value)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Name of parameter. |
| Parameter 3 | Return parameter for queried value. |
| Description | Returns the value of the parameter with the given name. |
| Return value | 0 : Value is queried data<br><br><0: Cannot query parameter |

| **Function** | **vaSi_CustomOp_GetInputImage** |
|---|---|
| Syntax | `int vaSi_CustomOp_GetInputImage (va_custom_op_sim_handle simHandle, const char* portName, va_image_handle *image)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Name of operator port. |
| Parameter 3 | Return parameter for image handle. |
| Description | Take an image which enters the operator at the given port and return a handle referencing that image. Before returning from the simulation function this image must either be stored by calling `vaSi_CustomOp_StoreImage()` or deleted by calling `vaSi_CustomOp_DeleteImage()`. |
| Return value | 0: OK<br><br><0 : Cannot get image |

| **Function** | **vaSi_CustomOp_PutOutputImage** |
|---|---|
| Syntax | `int vaSi_CustomOp_PutOutputImage (va_custom_op_sim_handle simHandle, const char* portName, va_image_handle imageHandle)` |

| **Function** | **vaSi_CustomOp_PutOutputImage** |
|---|---|
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Name of operator port. |
| Parameter 3 | Image handle. |
| Description | Outputs image to the given port. |
| Return value | 0 : Operation has been completed successfully<br><0: Cannot output image |

| **Function** | **vaSi_CustomOp_InputHasImage** |
|---|---|
| Syntax | `bool vaSi_CustomOp_InputHasImage (va_custom_op_sim_handle simHandle, const char* portName)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Name of operator input port. |
| Description | Returns whether there is an input image available at the port with the given name. |
| Return value | true : Image is available<br>false : No image available |

| **Function** | **vaSi_CustomOp_OutputReady** |
|---|---|
| Syntax | `bool vaSi_CustomOp_OutputReady (va_custom_op_sim_handle simHandle, const char* portName)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Name of operator output port. |
| Description | Returns whether the output port with the given name may take an image. |
| Return value | true : Output ready for next image<br>false : Output not ready for taking image |

| **Function** | **vaSi_CustomOp_CreateImage** |
|---|---|
| Syntax | `int vaSi_CustomOp_CreateImage (va_custom_op_sim_handle simHandle, va_image_format_handle format, va_image_handle * newImage)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Image format of the new image. |
| Parameter 3 | Return parameter for image handle. |
| Description | Creates a blank image based on the format given by parameter 2.<br>Before returning from the simulation function this image must either be stored by calling `vaSi_CustomOp_StoreImage()` or deleted by calling `vaSi_CustomOp_DeleteImage()`. |
| Return value | 0 : OK<br><0 : Could not create image |

| **Function** | **vaSi_CustomOp_DeleteImage** |
|---|---|
| Syntax | `int vaSi_CustomOp_DeleteImage (va_custom_op_sim_handle simHandle, va_image_handle imageHandle)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Handle of image which shall be deleted. |
| Description | Deletes image referenced by given image handle. |
| Return value | 0 : Operation has been completed successfully |

| Function | vaSi_CustomOp_DeleteImage |
|---|---|
| | <0: Error during deleting image |

| Function | vaSi_CustomOp_StoreImage |
|---|---|
| Syntax | `int vaSi_CustomOp_StoreImage (va_custom_op_sim_handle simHandle, va_image_handle imageHandle, const char* storeName)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Image handle. |
| Parameter 3 | Name as which the image shall be stored. The image may later be queried by this name. |
| Description | Stores image in local storage of the operator simulation instance. |
| Return value | 0 : Operation has been completed successfully<br>`VA_SIM_CANNOT_STORE_IMAGE`: Cannot create storage for image<br>`VA_SIM_STORE_NAME_ALREADY_USED`: Name “storeName” is already in use for currently stored image |

| Function | vaSi_CustomOp_GetStoredImagesCount |
|---|---|
| Syntax | `int vaSi_CustomOp_GetStoredImagesCount (va_custom_op_sim_handle simHandle, unsigned int *count)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Return parameter for image count. |
| Description | Returns the number of images which are stored within the operator simulation instance. |
| Return value | 0: OK<br><0: Can’t query information. |

| Function | vaSi_CustomOp_GetStoredImage |
|---|---|
| Syntax | `int vaSi_CustomOp_GetStoredImage (va_custom_op_sim_handle simHandle, unsigned int index, va_image_handle *retImage)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Index within the array of stored images. |
| Parameter 3 | Return parameter for image handle. |
| Description | Get image which has been stored before. The image is removed from the image storage. Before returning from the simulation function this image must either be stored again by calling `vaSi_CustomOp_StoreImage()` or deleted by calling `vaSi_CustomOp_DeleteImage()`. |
| Return value | 0 : OK<br><0 : Could not get image |

| Function | vaSi_CustomOp_GetNameOfStoredImage |
|---|---|
| Syntax | `const char* vaSi_CustomOp_GetNameOfStoredImage (va_custom_op_sim_handle simHandle, unsigned int index)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Index within the array of stored images. |
| Description | Returns a string of the image name. |
| Return value | Not `NULL` : Value is image name string<br>`NULL` : Could not query name |

| **Function** | **vaSi_CustomOp_GetStoredImageByName** |
|---|---|
| Syntax | `int vaSi_CustomOp_GetStoredImageByName (va_custom_op_sim_handle simHandle, const char* storeName, va_image_handle *retImage)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Name under which the image has been stored. |
| Parameter 3 | Return parameter for image handle. |
| Description | Get image which has been stored before with the given storage name. The image is removed from the image storage. Before returning from the simulation function this image must either be stored again by calling `vaSi_CustomOp_StoreImage()` or deleted by calling `vaSi_CustomOp_DeleteImage()`. |
| Return value | 0 : OK<br><0 : Could not get image |

| **Function** | **vaSi_CustomOp_CreateImageFormat** |
|---|---|
| Syntax | `int vaSi_CustomOp_CreateImageFormat (va_custom_op_sim_handle simHandle, const char* portName, va_image_format_handle* createdFormat)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Name of operator port. |
| Parameter 3 | Return pointer for format handle. |
| Description | Creates a new image format object and returns a corresponding handle. The format is initialized by the format of the port with the given name. Before returning from the simulation function the format must become deleted by calling `vaSi_CustomOp_DeleteImageFormat()`. |
| Return value | 0 : OK<br><0 : Could not create format |

| **Function** | **vaSi_CustomOp_CopyImageFormat** |
|---|---|
| Syntax | `int vaSi_CustomOp_CopyImageFormat (va_custom_op_sim_handle simHandle, va_image_format_handle formatHandle, va_image_format_handle *createdFormat)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Handle of format which is being copied. |
| Parameter 3 | Return parameter for format handle. |
| Description | Creates a new image format object and returns a corresponding handle. The format is initialized by the provided format. Before returning from the simulation function the format must become deleted by calling `vaSi_CustomOp_DeleteImageFormat()`. |
| Return value | Not `NULL` : Value is format handle<br>`NULL` : Could not create format |

| **Function** | **vaSi_CustomOp_DeleteImageFormat** |
|---|---|
| Syntax | `int vaSi_CustomOp_DeleteImageFormat (va_custom_op_sim_handle simHandle, va_image_format_handle formatHandle)` |
| Parameter 1 | Simulation handle provided to the operator simulation function. |
| Parameter 2 | Handle of format which is being deleted. |
| Description | Deletes the image format object referenced by the given format handle. |
| Return value | 0: OK |

| Function | vaSi_CustomOp_DeleteImageFormat |
|---|---|
| | <0 : Could not delete format |

| Function | vaSi_Image_GetFormat |
|---|---|
| Syntax | `int vaSi_Image_GetFormat (va_image_handle imageHandle, va_image_format_handle formatHandle)` |
| Parameter 1 | Image handle. |
| Parameter 2 | Handle of earlier created format which will be set to format of image. |
| Description | Queries the format of the image referenced by the image handle. |
| Return value | 0 : Operation has been completed successfully<br><0: Cannot query format |

| Function | vaSi_Image_SetProperty |
|---|---|
| Syntax | `int vaSi_Image_SetProperty (va_image_handle imageHandle, const char* propType, const va_data* propData)` |
| Parameter 1 | Image handle. |
| Parameter 2 | String identifying the property which shall be set. |
| Parameter 3 | Pointer to data structure which will be used for setting the new property. |
| Description | Set property of the image referenced by the image handle. Following properties may be set via this function:<br>`ImgWidth`: Set image width (`propData` has type `VA_UINT32`)<br>`ImgHeight`: Set image height (`propData` has type `VA_UINT32`) |
| Return value | 0 : Property has been set successfully<br>`VA_SIM_INVALID_PARAMETER`: Cannot identify property<br>`VA_SIM_INVALID_TYPE`: Property data has wrong format<br>`VA_SIM_INVALID_VALUE`: Property data has invalid value |

| Function | vaSi_Image_GetProperty |
|---|---|
| Syntax | `int vaSi_Image_GetProperty (va_image_handle imageHandle, const char* propType, va_data* propData)` |
| Parameter 1 | Image handle. |
| Parameter 2 | Enum value identifying the property which shall be queried. |
| Parameter 3 | Pointer to data structure which will be used for data communication. |
| Description | Queries the properties of the image referenced by the image handle. Following properties are available:<br>`ImgWidth`: Get image width (`propData` has type `VA_UINT32`)<br>`ImgHeight`: Get image height (`propData` has type `VA_UINT32`) |
| Return value | 0 : Property has been queried successfully<br>`VA_SIM_INVALID_PARAMETER`: Cannot identify property<br>`VA_SIM_INVALID_TYPE`: Property data has wrong format<br>`VA_SIM_INVALID_VALUE`: Property data has invalid value |

| Function | vaSi_Image_SetLineLength |
|---|---|
| Syntax | `int vaSi_Image_SetLineLength (va_image_handle imageHandle, unsigned int line, unsigned int length)` |

| Function | vaSi_Image_SetLineLength |
|---|---|
| Parameter 1 | Image handle. |
| Parameter 2 | Line number. |
| Parameter 3 | Line length. |
| Description | Sets the length of the referenced line to an individual value which may differ to the overall image width (not exceeding the maximum image width defined by the image format). |
| Return value | 0 : Operation has been completed successfully<br><0: Cannot set line length to the given value |

| Function | vaSi_Image_GetLineLength |
|---|---|
| Syntax | `int vaSi_Image_GetLineLength (va_image_handle imageHandle, unsigned int line, unsigned int *length)` |
| Parameter 1 | Image handle. |
| Parameter 2 | Line number. |
| Parameter 3 | Return parameter for line length. |
| Description | Returns the length of the referenced line. |
| Return value | 0: OK<br><0: Cannot query line length |

| Function | vaSi_Image_SetPixelValue |
|---|---|
| Syntax | `int vaSi_Image_SetPixelValue (va_image_handle imageHandle, uint64_t imagePos, unsigned int compIndex, int64_t value)` |
| Parameter 1 | Image handle. |
| Parameter 2 | Position within the frame. |
| Parameter 3 | Component index. |
| Parameter 4 | Pixel component value. |
| Description | Sets the corresponding pixel component to the given value. |
| Return value | 0 : Operation has been completed successfully<br><0: Error setting the pixel component value |

| Function | vaSi_Image_GetPixelValue |
|---|---|
| Syntax | `int vaSi_Image_GetPixelValue (va_image_handle imageHandle, uint64_t imagePos, unsigned int compIndex, int64_t *value)` |
| Parameter 1 | Image handle. |
| Parameter 2 | Position within the frame. |
| Parameter 3 | Component index. |
| Parameter 4 | Return parameter for pixel component value |
| Description | Returns the corresponding pixel component value. |
| Return value | 0 : Operation has been completed successfully<br><0: Error getting the pixel component value |

| Function | vaSi_ImageFormat_SetProperty |
|---|---|
| Syntax | `int vaSi_ImageFormat_SetProperty (va_image_format_handle formatHandle, const char* propType, const va_data* propData)` |
| Parameter 1 | Image format handle. |

| Function | vaSi_ImageFormat_SetProperty |
|---|---|
| Parameter 2 | Enum value identifying the property which shall be set. |
| Parameter 3 | Pointer to data structure which holds the new property. |
| Description | Sets properties of the image format referenced by the handle. Following properties may be set via this function:<br><br>Protocol: Set image protocol where *propData has the type VA_ENUM and is set to one of the following values:<br><br>• VALT_IMAGE2D<br><br>• VALT_LINE1D<br><br>ColorFormat: Set image protocol where *propData has the type VA_ENUM and is set to one of the following values:<br><br>• VAF_GRAY<br><br>• VAF_COLOR<br><br>ColorFlavor: Set image protocol where *propData has the type VA_ENUM and is set to one of the following values:<br><br>• FL_NONE<br><br>• FL_HSI<br><br>• FL_YUV<br><br>• FL_LAB<br><br>• FL_RGB<br><br>• FL_XYZ<br><br>Parallelism: Set parallelism (type VA_INT32)<br><br>ComponentCount: Set number of pixel components (type VA_INT32)<br><br>ComponentWidth: Set pixel component width (type VA_INT32)<br><br>Arithmetic: Set pixel component arithmetic where *propData has the type VA_ENUM and is set to one of the following values:<br><br>• UNSIGNED<br><br>• SIGNED<br><br>MaxImgHeight: Set max. image height (type VA_INT32)<br><br>MaxImgWidth: Set max. image width (type VA_INT32) |
| Return value | 0 : Property has been set successfully<br><br>VA_SIM_INVALID_PARAMETER: Cannot identify property<br><br>VA_SIM_INVALID_TYPE: Property data has wrong format<br><br>VA_SIM_INVALID_VALUE: Property data has invalid value |

| Function | vaSi_ImageFormat_GetProperty |
|---|---|
| Syntax | int vaSi_ImageFormat_GetProperty (va_image_format_handle formatHandle, VAImageFormatProperty propType, va_data* propData) |

| Function | vaSi_ImageFormat_GetProperty |
|---|---|
| Parameter 1 | Image format handle. |
| Parameter 2 | Enum value identifying the property which shall be queried. |
| Parameter 3 | Pointer to data structure which will be overwritten by the queried property. |
| Description | Queries properties of the image format referenced by the handle. The properties which may be queried are identical to the ones which can be set through the function `vaSi_ImageFormat_SetProperty()`. |
| Return value | 0 : Property has been queried successfully<br>`VA_SIM_INVALID_PARAMETER`: Cannot identify property<br>`VA_SIM_INVALID_TYPE`: Property data has wrong format |

| Function | vaSi_CreateStatusMessage |
|---|---|
| Syntax | `int vaSi_CreateStatusMessage (va_custom_op_sim_handle simHandle, va_status_handle *newMessage)` |
| Parameter 1 | Simulation handle. |
| Parameter 2 | Return parameter for created error message. |
| Description | Create a status message which may be submitted to the simulation engine. |
| Return value | 0 : OK<br><0: Can’t create message |

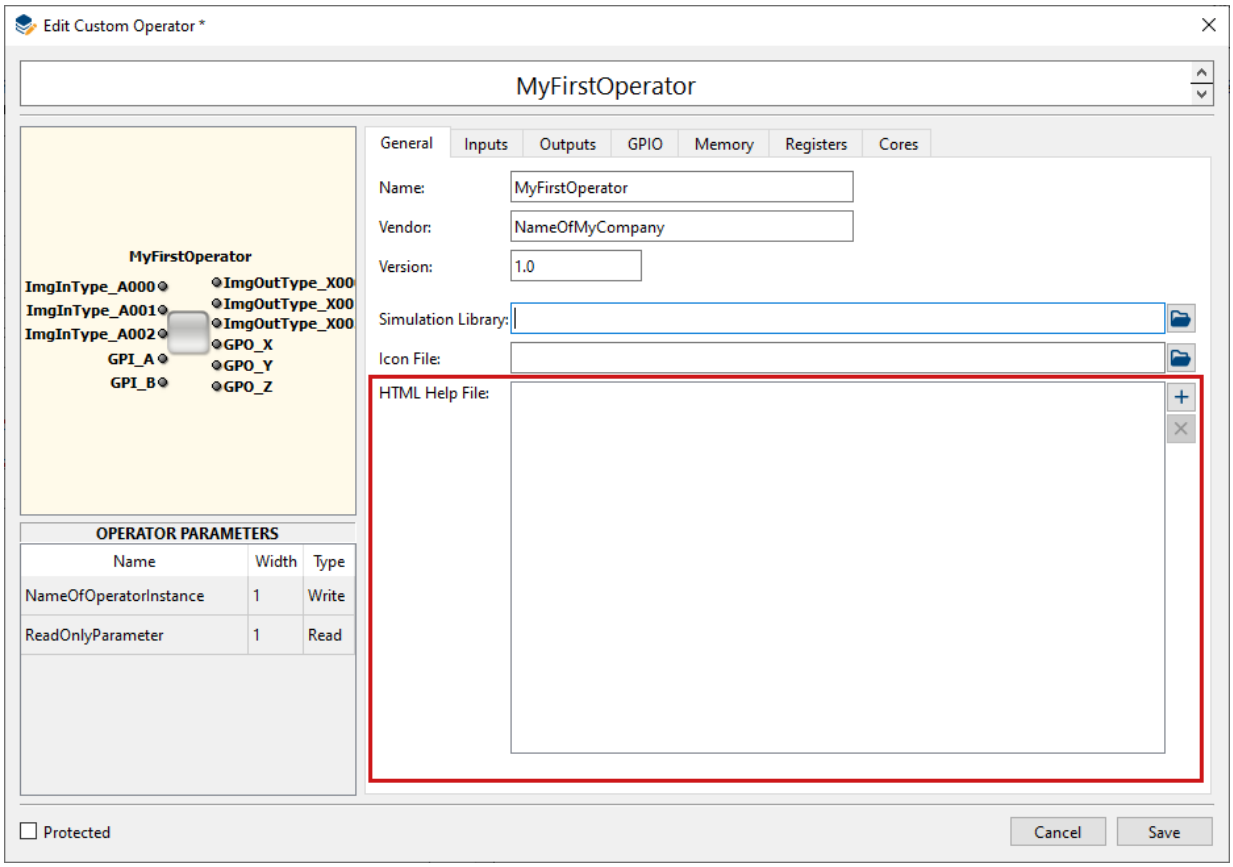
| Function | vaSi_SetStatusMessageProperty |
|---|---|
| Syntax | `int vaSi_SetStatusMessageProperty (va_custom_op_sim_handle simHandle, va_status_handle message, const char* propName, const va_data* propValue)` |
| Parameter 1 | Simulation handle. |
| Parameter 2 | Status message handle. |
| Parameter 3 | Name of property which shall be set. |
| Parameter 4 | New property value |
| Description | Alter status message property. Following properties may be set via this function:<br>`Code`: Set error code (type `VA_INT32`)<br>`Severity`: Set severity level where the data (type `VA_ENUM`) must be one of the following values:<br>• `VA_INFO`<br>• `VA_WARNING`<br>• `VA_ERROR`<br>`Description`: Set string description of status (type `VA_STRING`) |

| Function | vaSi_SendStatusMessage |
|---|---|
| Syntax | `int vaSi_SendStatusMessage (va_custom_op_sim_handle simHandle, va_status_handle message)` |
| Parameter 1 | Simulation handle. |
| Parameter 2 | Handle of status message which shall be submitted. |
| Description | Submitted status message to the simulation engine. |

| Function | vaSi_SendStatusMessage |
|---|---|
| Return value | 0 : OK<br><0: Can't submit message |

## 4.3.13. Creating Custom Operator Documentation

Documentation of the operator should be provided as an HTML file. When available, all files which make up the documentation need to be specified under tab **General** -> **HTML Help Files**:




The first file that is specified is interpreted to be the starting point of the operator's documentation. The naming convention for this file is: **<NameOfCustomOperator>**`.htm`.

**Make sure you provide a CSS file.** Make sure you also provide all related image files.

You can use the operator template provided in the VisualApplets install directory in subdirectory `Examples/CustomLibrary/OperatorTemplate`.

## 4.3.14. Completing the Custom Operator

When you have wrapped your HDL code so that its interface matches the generated black box, you need to proceed some last steps for completing your custom operator:

1. Create a netlist out of your implementation.



**Set Add IO Buffer = NO**

When creating the netlist, make sure that your synthesis tool doesn't automatically add IO buffers. In case you use XST for netlist synthesis you set

```
Add IO Buffer = NO
```

Otherwise, the resulting NGC file will cause errors during the VisualApplets build flow.



**Warnings During Netlist Generation**

When generating the net list, warnings may be output concerning unused IO Ports of the custom operator interface. Unused IO Ports are all ports that were generated according to your operator definition, but are not connected with your IP core. You may ignore these warnings.

Examples of this behavior are all custom operator examples you find in the `Examples` directory of your VisualApplets installation:

`\Examples\CustomLibrary`

2. If required, also define a constraints file (`*.ucf` format if you use Xilinx ISE, `*.xdc` format if you use Xilinx Vivado).

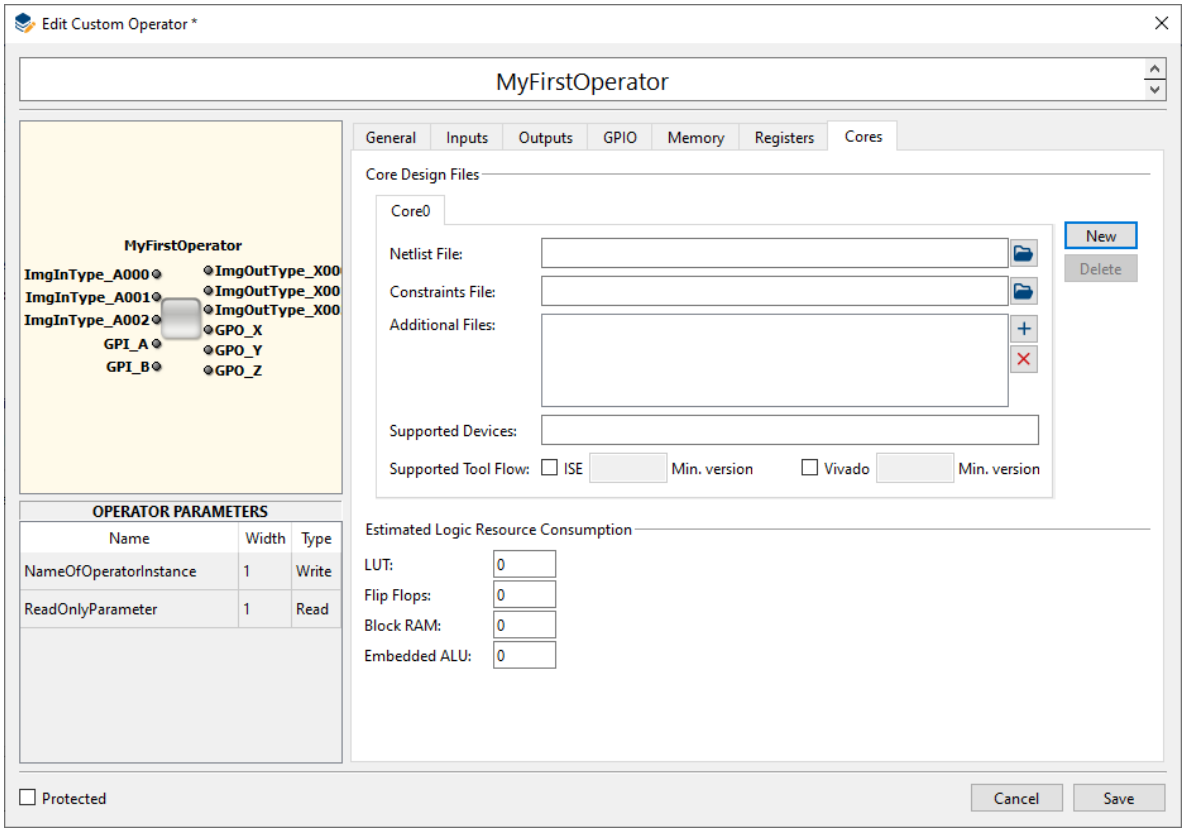
3. Optionally, set up the operator's software interface as described in section Section 4.3.10, 'Defining the Custom Operator's Software Interface'.

4. Optionally, create the operator documentation as described in section Section 4.3.13, 'Creating Custom Operator Documentation'.

Now, you need to complete the operator definition in VisualApplets. To do so, proceed as follows.

Required steps:

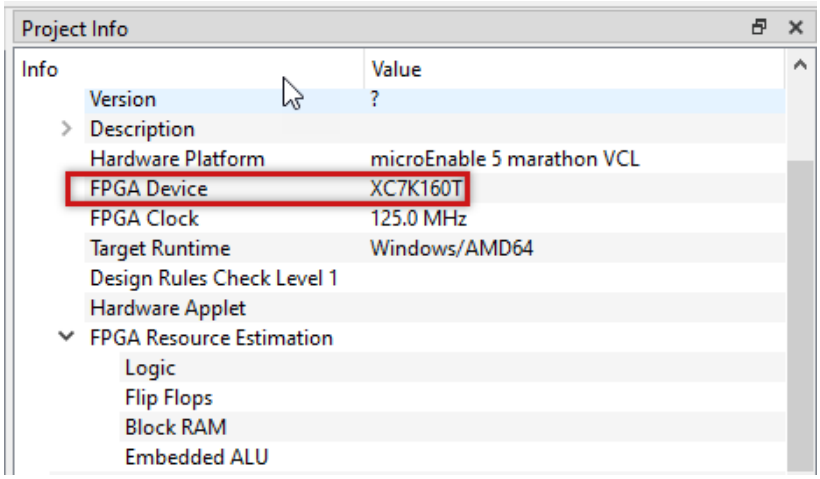
1. Go to the **Cores** tab.




2. Specify the netlist file you generated.

3. Specify the constraints file if you defined constraints.

4. Specify the supported devices: The device name is the name of the FPGA type of the target platform. Please use exactly the same spelling as provided in the project info box of VisualApplets. If several FPGA types are supported, use a space separated list of names.




   If a design uses the custom operator, but the FPGA on the target platform is not in this list, the DRC will report an error that the operator is not supported by the target platform.

5. Specify the supported Xilinx Tool(s). You can check the boxes for both ISE and Vivado. Netlists generated with ISE are usually also compatible with the Vivado build flow but you should check whether this is the case for your operator implementation. You need to define the minimal version number of the tool which supports the given netlist. Typically this would be the version which you used for creating the net list.

   If a design uses the custom operator, but the specified tools are not used for building the design, the DRC will report an error that the operator is not supported by the target platform.



**Defining Multiple Cores**

You can define multiple cores for the same custom operator. This will allow to use device and tool specific implementations of the custom operator so for different target platforms the appropriate implementation is chosen for building an applet.

   Optional steps:

6. Optionally, enter the consumption of logic resources by the operator. Simply enter the values estimated by the Xilinx tools during generation of netlist.

7. Under the **General** tab, specify the path to your simulation library (the custom operator's software interface).

8. Under the **General** tab, specify the path to the icon file. This is the file that contains the icon that will be used when your custom operator is displayed in VisualApplets.

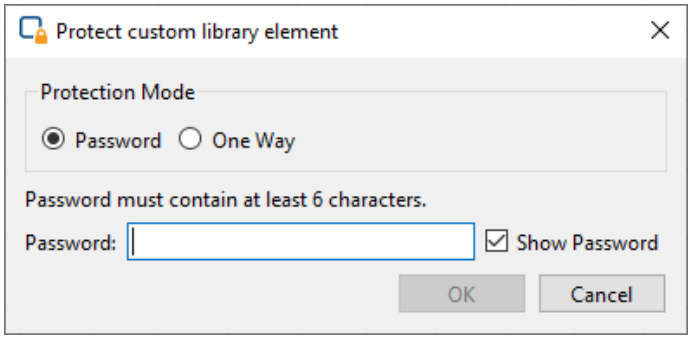
9. Under the **General** tab, specify all files that make up your custom operator documentation. Make sure you also provide a CSS file and all related image files.

10. If you want to protect your operator design: In the left bottom corner, activate the option **Protected**. In the dialog that opens:

    a. Make sure protection mode **Password** is activated.

    b. Enter your password.

    c. Click **OK**.




You can always protect your custom operator design also at a later point of time, using the context menu of the custom library element.

**Protecting Options**

After protection has been enabled, the custom operator is made a "black box". There are two ways to protect a custom operator design:

- **Protection via password**: The custom operator design can afterwards be opened and edited via password. Users that do not have the password will not be able to see any details of the custom operator (black box).
- **Irreversible protection**: If you select protection mode **One-Way**, the custom operator is made a black box forever and cannot be re-opened, not even by yourself.

**"One-Way" protection is irreversible:** If you select protection mode **One Way** (instead of **Password**), the custom library element can never be re-opened, not even by yourself. If you plan to enhance the element at a later point of time, make sure you select protection mode **Password** instead. Alternatively, you can save a copy of the element (as a hierarchical box or a non-protected operator) before enabling this protection mode.

11. Click **Save**.

Now, your new custom operator is ready for being used in designs.

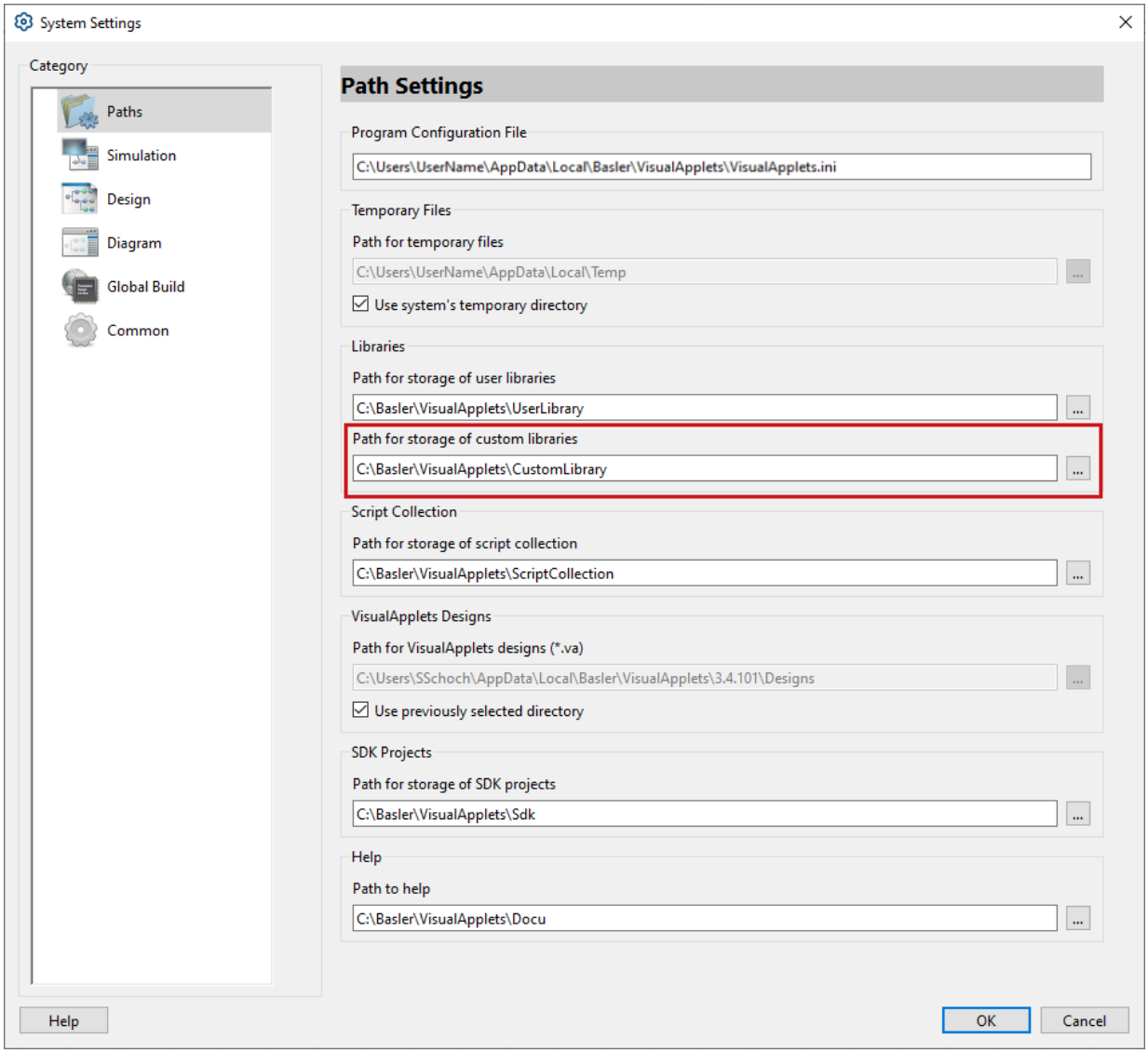
## 4.3.15. Using New Custom Operators

### 4.3.15.1. Distributing the Custom Library or the Individual Custom Operator

A custom library with all contained operators is stored as one single `<LibaryName>.val` or `<LibaryName>.vl` file.

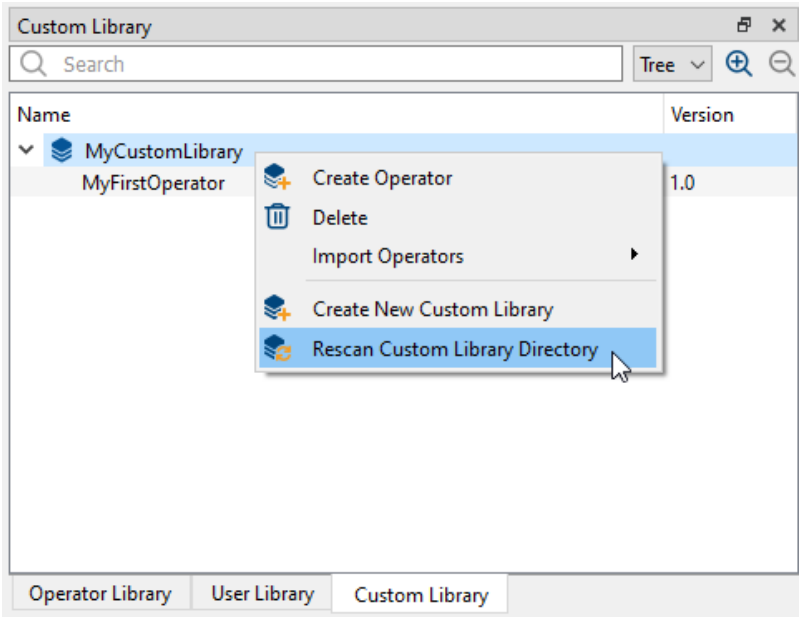
`<LibaryName>` is the name of the custom library.

This file can be distributed and directly applied in VisualApplets. It simply needs to be copied into the Custom Library directory which is specified in the VisualApplets settings ( **Settings** -> **System Settings** -> **Paths** -> **Custom Libraries**




1. Copy the new `<LibaryName>.val` or `<LibaryName>.vl` file to the Custom Library directory of your VisualApplets installation.

2. Re-scan the custom library in the VisualApplets GUI: Right-click on the library name and from the sub menu select **Rescan Custom Library Directory**.

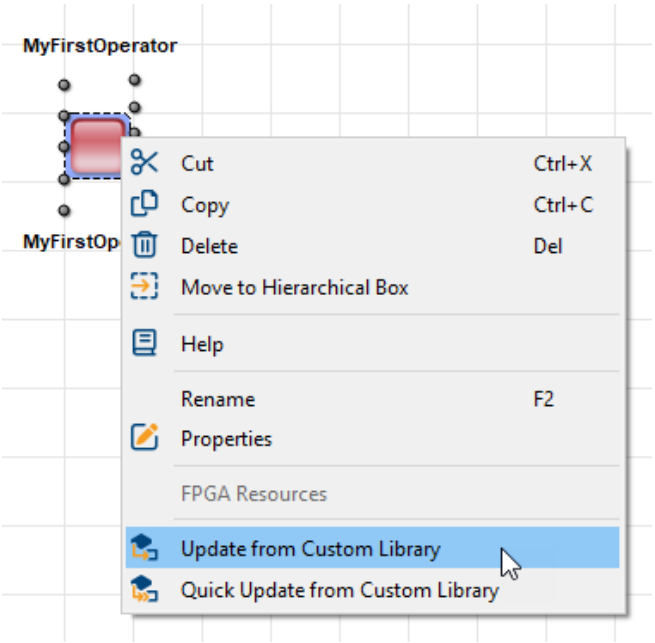



In the VisualApplets examples directory, you find a ready-to-use library called `CustomLibrary.vl` which contains all example operators.

### 4.3.15.2. Update from Custom Library

When you make changes to a custom operator, these changes are not reflected in the designs where you already use the custom operator. Therefore, you need to update the custom operator instances in the designs.

1. Right-click on the operator.
2. From the sub-menu, select **Update from Custom Library** or **Quick Update from Custom Library**.



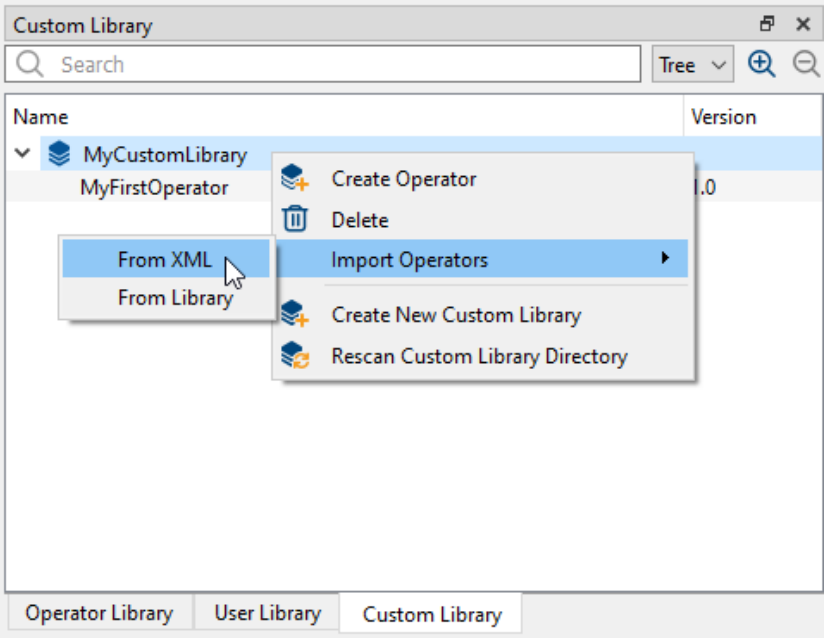

The update mechanism for Custom Libraries is exactly the same as for User Libraries.

### 4.3.15.3. Importing and Exporting Individual Custom Operators

You can import and export individual custom operators by importing/exporting the XML definition of the operator.

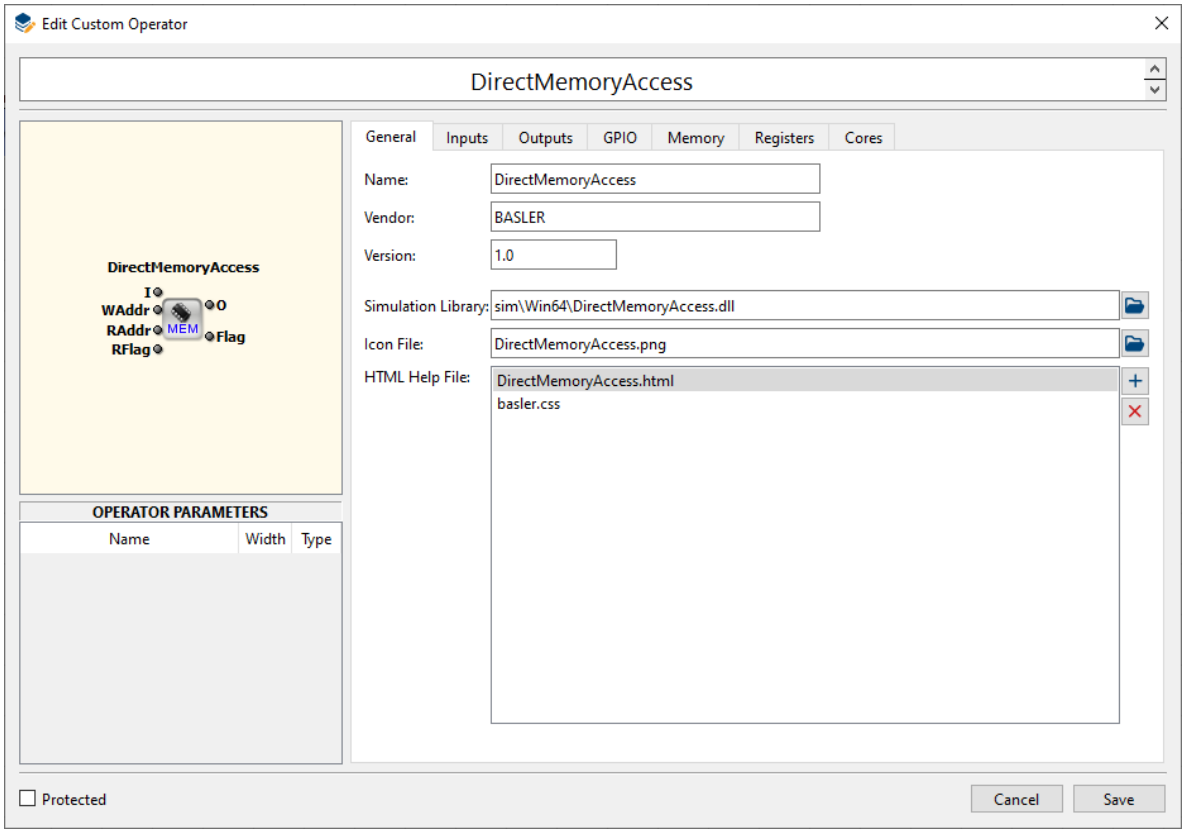
To import a custom operator:

1. Right-click on the custom library where you want to import the custom operator to.
2. From the sub-menu, select **Import Operator** -> **From XML**.



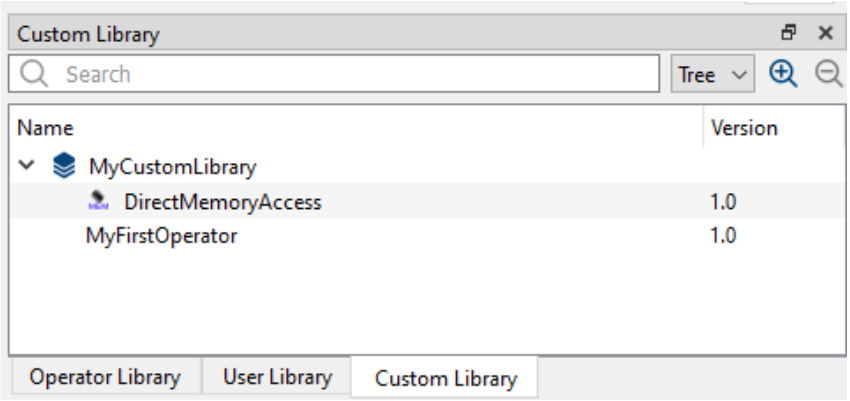

3. Specify the path to the custom operator's XML definition and click **Open**.

   Immediately, the **Edit Custom Operator** dialog opens:




4. Click **Save**.

   After saving, the imported operator is directly available in the custom library:




## 4.3.16. Operator Template and Examples

### 4.3.16.1. Examples

In the install directory, you find three completed custom operators which you can use as reference. You find the examples here:

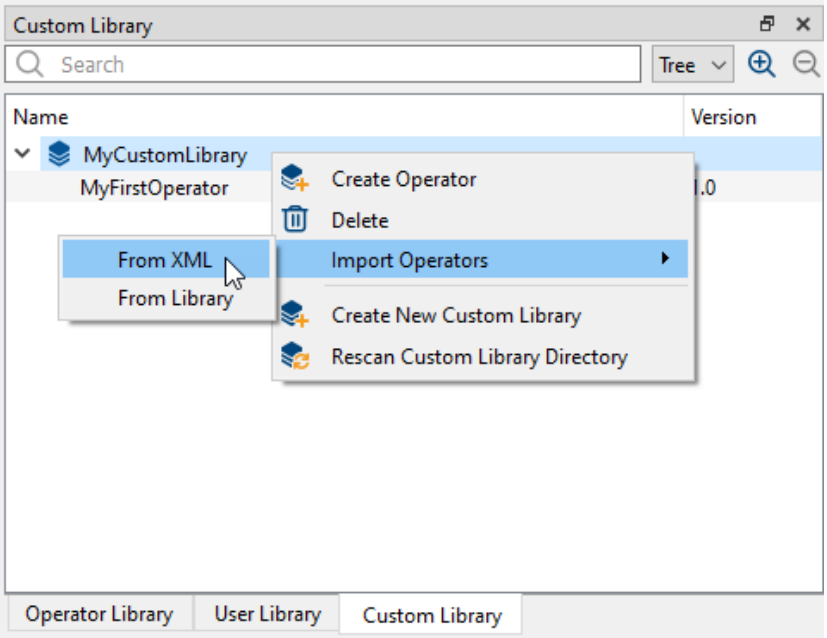
`\Examples\CustomLibrary`

### 4.3.16.2. Custom Operator Template

In the install directory, you find a custom operator template which you can use for defining your custom operators.

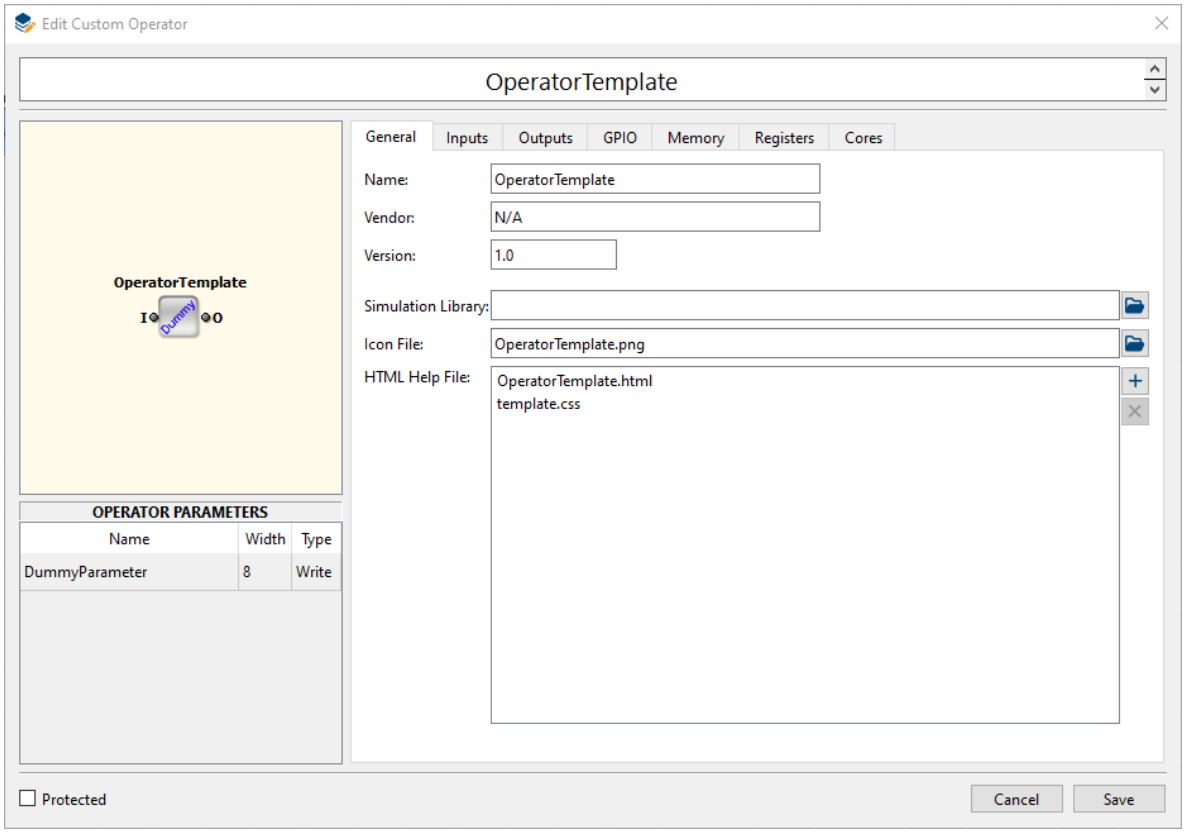
`\Examples\CustomLibrary` To use the custom operator template:

1. Right-click on the custom library where you want to create the new custom operator in.
2. From the sub-menu, select **Import Operator** -> **From XML**.




3. Specify the path to the operator template.
4. Click **Open**.

   Immediately, the **Edit Custom Operator** dialog opens:




5. Give a name to your new custom operator and proceed as described in section Section 4.3.6, 'Defining an Individual Custom Operator via GUI'.

## 4.3.17. XML Format for Custom Operator Specification

The definition of a custom operator is stored in XML format. A concerning XML file can be exported from the operator library or an operator can be imported using an earlier exported XML file.

In the following, we describe the required parameters where the parameter name is related to an XML tag with the same name. A parameter like `ImgInInfo` will translate to an XML entry like:

`<ImgInInfo> ImgInPortNames </ImgInInfo>` where `ImgInPortNames` is the value which in this case would be a sequence of port names. The parameters are hierarchically ordered. In the following tables, lines with gray background will notify the hierarchy position where the parameters are expected.

Simple parameter values can be of following types:

- Choice: the allowed values are `YES` or `NO`
- String: an ASCII string without whitespace
- Integer
- Floating-point

Some parameters are composed as a structure of values where arrays or records are possible elements for structuring. Arrays are entered by a list of values separated by white space where the values themselves may be structured. Records are entered by a scheme like follows where `RecordName` is the record identifier, `attrX` are the identifiers for the record entries and `attrXValue` are the values:

```
<RecordName attr1=”attr1Value” .. attrN=”attrNValue”/>
```

An example would be providing a record called `port` with entries for name and width:

```
<port name=”flag” width=”4”/>
```

The root tag of the XML format is *Operator* with an attribute `name` where the custom operator name should be provided:

```
<Operator name=”XYZ”>
…
</Operator>
```



**Comply with VHDL Naming Conventions**

When defining the operator name in the VA GUI, make sure you conform to the VHDL naming conventions.

VHDL valid names are defined as follows:

“A valid name for a port, signal, variable, entity name, architecture body, or similar object consists of a letter followed by any number of letters or numbers, without space. A valid name is also called a named identifier. VHDL is not case sensitive. However, an underscore may be used within a name, but may not begin or end the name. Two consecutive underscores are not permitted.”

| Parameter Name | Type | Description |
|---|---|---|
| `Vendor` | String | Name of Vendor. |
| `Version` | String | Version number of the operator. The value can be freely chosen and is intended for version identification by the user. |
| `Cores` | Array of String | List of core netlists for the operator. The first string must be `Core0` and must always be there. If more than one core is available the naming convention for them is `Core`**<N>** where **<N>** is a integer number incremented with every core. |
| `LibraryFile` | String | Quoted name of file containing software library (dynamic link library) containing the high-level simulation model for the operator. |
| `IconFile` | String | Quoted name of file containing the operator icon. |
| `HtmlHelpFiles` | Array of String | List of quoted file names which contain help content (html + images). The first file is considered as the main HTML file. |

Table 4.1. Operator/Info

| Parameter Name | Type | Description |
|---|---|---|
| `RegInInfo` | Array of String | List of names of later defined info structures (`Operator/RegIn`) describing write register ports. |
| `RegOutInfo` | Array of String | List of names of later defined info structures (`Operator/RegOut`) describing read register ports. |
| `ImgInSyncMode` | String | String defining whether the inputs at the **ImgIn** ports are synchronous or asynchronous to each other. This string may either be “Sync” or “Async”. |
| `ImgInInfo` | Array of String | List of names of later defined info structures (`Operator/ImgIn`) describing the properties of |

| Parameter Name | Type | Description |
|---|---|---|
| | | the image input ports. Several list entries may refer to the same structure which then means that several ports of the same kind of image input interface are available. |
| ImgOutInfo | Array of String | List of names of later defined info structures (Operator/ImgOut) describing the properties of the image output ports. Several list entries may refer to the same structure which then means that several ports of the same kind of image output interface are available. |
| GPIn | Array of String | List of pin names for general purpose signal inputs. |
| GPOut | Array of String | List of pin names for general purpose signal outputs. |
| MemInfo | Array of String | List of names of later defined info structures (Operator/Mem) describing the properties of the memory interface ports. Several list entries may refer to the same structure which then means that several ports of the same kind of memory interface are available. |

Table 4.2. Operator/IO

| Parameter Name | Type | Description |
|---|---|---|
| NrLut | Integer | Number of FPGA LUT elements consumed by the operator |
| NrRegs | Integer | Number of FPGA registers consumed by the operator |
| NrBlockRam | Integer | Number of block ram elements consumed by the operator |
| NrEmbeddedMult | Integer | Number of embedded multipliers consumed by the operator |

Table 4.3. Operator/Properties

Image input port specification is done by following syntax within the configuration file:

```
<ImgIn name="IMG_IN_IDENTIFIER"> Parameters </ImgIn>;
```

Here IMG_IN_IDENTIFIER is one of the image input port names which have been provided in the above parameter Operator/IO/ImgInInfo. The content Parameters is specifying the properties of the image interface port:

| Parameter Name | Type | Description |
|---|---|---|
| Width | Integer | Width of the image data port |
| FIFODepth | Integer | Depth of the buffer FIFO for input data which at least needs to be provided by the VA core. The value must be a power of two minus 1 between 15 and 1023. |
| Formats | Array of Record | List of image format records ImgFormat which are supported by the port. For the naming scheme of image formats see below. |

Table 4.4. Operator/ImgIn

The image format records have the following structure:

```
<ImgFormat name="FORMAT" maxWidth="X1" maxHeight="Y1" alias="NAME1"/>
```

The entry FORMAT is a String value for an image format coded by the below discussed naming scheme for image formats. The attributes maxWidth and maxHeight are optional and fix the limits of image size. If they are not present, the image size constraints can be freely chosen by the user within VisualApplets later on. The attribute alias is optional as well and, if present, defines the name under which the format will be displayed in the GUI.

Image output port specification is done by following syntax within the configuration file:

```
<ImgOut name="IMG_OUT_IDENTIFIER"> Parameters </ImgOut>;
```

Here IMG_OUT_IDENTIFIER is one of the image output port names which have been provided in the above parameter Operator/IO/ImgOutInfo. The content Parameters is specifying the properties of the image interface port:

| Parameter Name | Type | Description |
|---|---|---|
| Width | Integer | Width of the image data port |
| FIFODepth | Integer | Depth of the buffer FIFO for output data which at least needs to be provided by the VA core. The value must be a power of two minus 1 between 15 and 1023. |
| Formats | Array of Record | List of image format records ImgFormat which are supported by the port. For the naming scheme of image formats see below. |

Table 4.5. Operator/ImgOut

Image formats are coded by the following naming scheme:

```
{BaseFormat}{BitsPerPixel}x{Parallelism}
```

Optionally there can be suffixes for image dimension and the notification of signed component data:

```
{BaseFormat}{BitsPerPixel}x{Parallelism}x{Dimension}{Sign}
```

The meaning of the dimension is as follows:

- Dimension = 2 – a two-dimensional image means that the image is structured both by end-of-line and end-of-frame markers.
- Dimension = 1 – a one-dimensional image means that there are no end-of-frame markers which divide the incoming lines into frames.

When no dimension is specified a value of two is assumed. The suffix {Sign} can be s for signed pixel components or u for unsigned values where the default value is u when no such suffix is provided. Supported color formats are **rgb**, **yuv**, **hsi**, **lab** and **xyz**.

Examples are:

- **gray8x4** – gray format with 8-bit pixel and parallelism 4
- **rgb24x2** – rgb color format with 3x8-bit pixel and parallelism 2
- **gray16x1** – gray format with 16-bit pixel, only single pixel in a data word
- **gray8x4x1** – one dimensional gray image with 8-bit per pixel and parallelism 4
- **gray16x1s** – gray image with signed 16-bit components, only single pixel in a data word

Register input port specification is done by following syntax within the configuration file:

```
<RegIn name="REG_IN_IDENTIFIER"> Parameters </RegIn>;
```

Here REG_IN_IDENTIFIER is one of the register input port names which have been provided in the above parameter Operator/IO/RegInInfo. The content Parameters is specifying the properties of the register interface port:

| Parameter Name | Type | Description |
|---|---|---|
| Width | Integer | Width of the register port |

Table 4.6. Operator/RegIn

Register output port specification is done by following syntax within the configuration file:

<RegOut name="REG_OUT_IDENTIFIER"> Parameters </RegOut>;

Here REG_OUT_IDENTIFIER is one of the register output port names which have been provided in the above parameter Operator/IO/RegOutInfo. The content Parameters is specifying the properties of the register interface port:

| Parameter Name | Type | Description |
|---|---|---|
| Width | Integer | Width of the register port |

Table 4.7. Operator/RegOut

Memory interface specification is done by sections with following syntax within the configuration file:

<Mem name="MEM_IDENTIFIER"> Parameters </MEM>

Here MEM_IDENTIFIER is one of the memory port names which have been provided in the above described parameter Operator/IO/MemInfo. The content Parameters is specifying the properties of the memory interface:

| Parameter Name | Type | Description |
|---|---|---|
| DataWidth | Integer | Data width |
| AddrWidth | Integer | Address width |
| WrFlagWidth | Integer | Width of flag for marking write accesses. This parameter must be >= 1. |
| RdFlagWidth | Integer | Width of flag for marking read accesses. This parameter must be >= 8. |
| WrCntWidth | Integer | Width of port for communicating the number of available write commands |
| RdCntWidth | Integer | Width of port for communicating the number of available read commands |
| SyncMode | String | This parameter signals the relation of the memory interface clock and the design clock. Following values are possible:<br><br>SyncToDesignClk – memory interface ports are synchronous to iDesignClk.<br><br>SyncToDesignClk2x – memory interface ports are synchronous to iDesignClk2x. |

Table 4.8. Operator/Mem

Specification of IP core netlists is done by sections with following syntax within the configuration file:

<Core name="CORE_IDENTIFIER"> Parameters </Core>

Here Core_IDENTIFIER is one of the core names which have been provided in the above described parameter Operator/Cores. The content Parameters is specifying the properties of the IP core:

| Parameter Name | Type | Description |
|---|---|---|
| Devices | Array of String | List of FPGA device names which are supported by the core (Example: XC3S1600E XC3S4000). |
| NetlistFile | String | Quoted UTF-8 encoded file name for the net list. |
| ConstraintsFile | String | Quoted UTF-8 encoded file name for an optional constraints file. |
| MinVersionISE | String | Minimum version number of ISE tool flow which can use the given netlist (Example: 14.6 for ISE 14.6). If ISE is not supported this string is empty. |
| MinVersion | String | Minimum version number of Vivado tool flow which can use the given netlist (Example: 2014.4 for Vivado 2014.4). If Vivado is not supported this string is empty. |

Table 4.9. Operator/Core

## 4.4. Multiple Processes

VisualApplets projects may include multiple processes. A process is the top level of a diagram and can include an arbitrary number of operators and *hierarchical boxes*. Each process can be started and stopped individually.

A good example where two processes are used is a two camera application. Have a look at the following figure. It shows two processes with a camera, a buffer and a DMA operator in each of the processes.

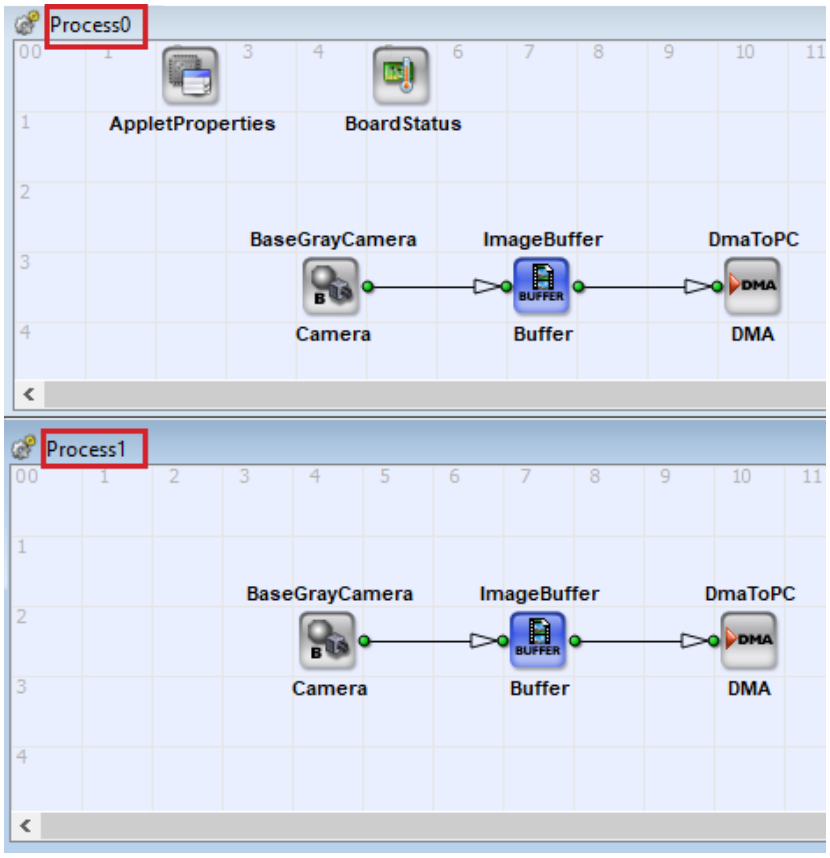



Figure 4.24. Applet with Two Processes

When the applet is used, both processes, i.e., both cameras can be started and stopped individually. If both cameras are in the same process, they have to be started and stopped simultaneously, as all DMAs of a process have always to be started.

To learn more about the usage of VisualApplets applets on hardware, see section Section 2.2, 'Running Your Applet on Hardware', and/or the Framegrabber SDK documentation [https://docs.baslerweb.com/frame-grabbers/managing-applets-micro-diagnostics].

### 4.4.1. Managing Processes

To add a new process, click **Design** -> **New Process** or choose the icon in the *Edit* toolbar.

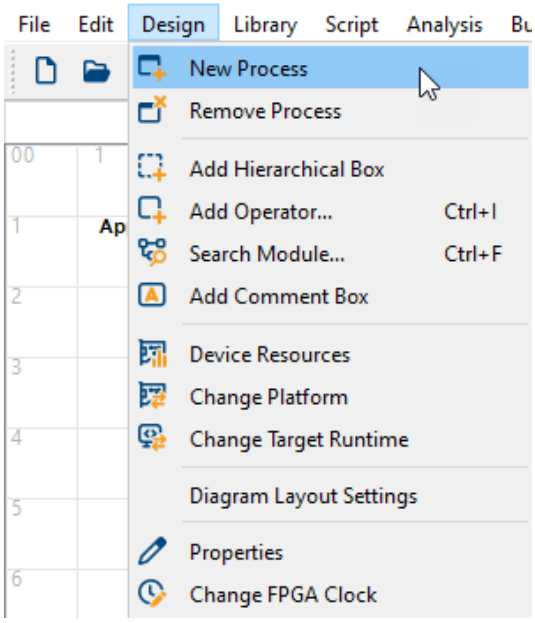



Figure 4.25. Creating a New Process

To remove a process first select any digram window belonging to the process and afterwards select **Design** -> **Remove Process** or choose the icon in the *Edit* toolbar.

Processes always start with index 0 and have to be in ascending order. If a process is removed, processes with higher indices are shifted to fill the gap.

### 4.4.2. Processes without DMAs / Trigger Processes

As mentioned, an applet process can only be used on the hardware after all DMAs of the respective process have been started. However, if a process does not contain any DMA channels, it will be immediately started after initialization of the applet. This is useful for trigger and signal processing applications as they are used before the image acquisition is started.

To learn more about initialization and usage of VisualApplets applets on hardware, see section Section 2.2, 'Running Your Applet on Hardware', and/or the Framegrabber SDK documentation [https://docs.baslerweb.com/frame-grabbers/managing-applets-micro-diagnostics].

### 4.4.3. Process Intercommunication

Image data between processes cannot be interchanged. However, using operators *TxSignalLinks* and *RxSignalLinks*, an interprocess communication for signals can be established.

## 4.5. Target Hardware Porting

A VisualApplets project is always based on a hardware platform, such as the microEnable IV VD4-CL, microEnable 5 VQ8-CXP6D etc. VisualApplets offers a powerful hardware porting system. The hardware platform can simply be changed. VisualApplets performs the porting to the new FPGA, DMA interfaces, memory, camera ports, etc. automatically. To perform a hardware porting open the design and select **Design** -> **Change Platform** from the main menu. VisualApplets will then ask for the new file name and allows the selection of the new hardware platform. A design rule check and the build of the new design file can immediately be performed.

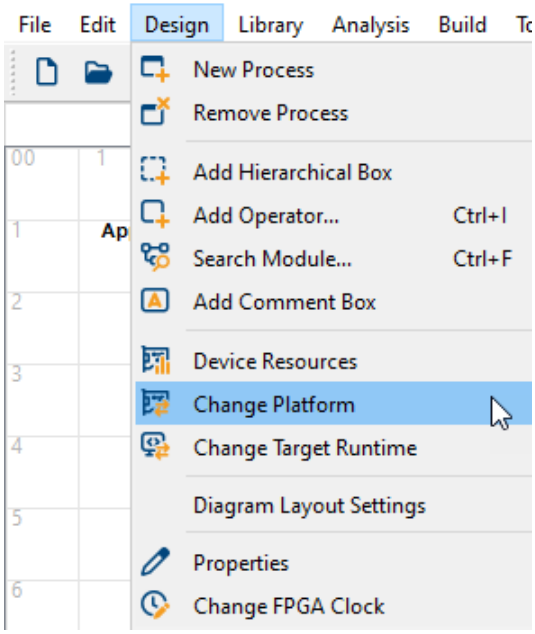



Figure 4.26. Target Hardware Porting

**Limitations**

Of course, the new platform has to be capable of implementing the design. In the following cases, you may need to adapt your design to match the new platform:

- A smaller FPGA might result in a logic overmap.
- A project cannot be ported if the device resources are not available. For example, the number of DMA channels or digital output ports might not be available on the new platform.
- Not all operators from the *Platform* libraries exist for each platform. For example, a Camera Link camera operator cannot be ported to a gigabit Ethernet camera operator. In this case, the missing operator is replaced by a *Dummy* operator.
- Also operators from other libraries (e.g., EventToHost) may not be available for the new platform. In this case, DRC1 delivers an according error message:




Figure 4.27. Error message in case an operator is not applicable for new hardware platform

## 4.6. PixelPlant Designs

The PixelPlant hardware is an add-on board family for microEnable IV. At the moment, two boards are available: PixelPlant100 and PixelPlant200 which differ in FPGA and RAM size.

### 4.6.1. PixelPlant Projects

To develop an applet for using the PixelPlant Px100 / Px200 hardware, two separate design parts have to be created:

1. A base design for the microEnable IV board
2. A design for PixelPlant

Both parts need to be handled in two separate VisualApplets design files. Both designs, PixelPlant and microEnable IV, have to be build, so that two HAP files are generated. Finally, both parts (HAP files) have to be merged into one hardware applet (HAP file).

For data transfer between the microEnable IV applet and the PixelPlant applet, operators *TxLink* and *RxLink* have to be used. *TxLink* operators transmit the data to a PixelPlant board, *RxLink* operators receive data from the PixelPlant. The PixelPlant part has to contain the corresponding *RxLink* and *TxLink* operators to pick the data from the microEnable IV and transmit it back. Each *RxLink* / *TxLink* operator has a channel ID parameter which selects the corresponding *RxLink* / *TxLink* operator on microEnable IV. Theses channel ID parameters as well as the link formats have to match on both, PixelPlant and microEnable IV.

The merging of two applets is explained in the next section.

### 4.6.2. HAP Merger

Merging PixelPlant and microEnable IV designs can be done by using the HAP Merger utility. Access it by **Build** -> **Hardware Applet Merger**. The utility dialog asks for the microEnable IV HAP file and its corresponding PixelPlant HAP file. Pressing the "Merge" button generates a HAP file containing both design parts. This merged HAP file is used to be loaded on the frame grabber.

Please note: The channel ID of corresponding RxLink / TxLink pairs have to match on both applets as well as the link formats. The hardware applet merger will not merge the HAPs if a mismatch is detected.

## 4.7. System Settings

Even if VisualApplets comes along with a useful basic configuration, it can be adapted individually to fit the user's needs in the best way. The configuration of the program can be changed and stored on the local computer. You can change your settings in several setting dialogs which are accessible over the user interface of VisualApplets.

You can define global settings for all users, personal settings for the current user, or local settings for a certain installation.

All configurations are stored in one configuration file (`VisualApplets.ini`). During the installation process, you have already selected the location of this file.

The following sections describe the setting options.

You can access the dialogs where you can specify your system settings as follows:

1. Select menu item **Settings** -> **System Settings**.

The *System Settings* window opens where you can choose between different setting categories. For each category, a settings dialog is displayed.

### 4.7.1. Path Settings

This dialog allows configuring the default locations of different types of files.



**Default Path Settings**

If Visual Applets is installed on your system in a folder that requires administrator rights (e.g., C:\Programs), you have to start VisualApplets with administrator rights in order to use the default paths.

Alternatively, you can adapt the path settings under *System Settings / Path Settings*. Also adapt the path under Section 4.7, 'System Settings' / *Global Build Settings / Generation of Hardware Applets / Path for storage of hardware applets (*.hap)* in this case.

**Temporary files**

*Path for temporary files:* While running, VisualApplets needs to generate temporary files. Usually, these files will be deleted when VisualApplets is closed. Select the directory where these files should be created.

*Use the system's temporary directory*: If you check this option, VisualApplets retrieves this location from the operating system. This is the recommended setting. If you want to use another directory for this purpose, un-check this box, specify an alternative directory in the field above and make sure it is not write-protected.

**User libraries**

*Path for storage of user libraries:* User libraries are containers for modules created by the user himself. (See the according chapter for further details.) In this field you can specify the directory where the user-defined modules are to be saved to and loaded from. Change this directory to another directory (e.g., to one on a network drive) when you are sharing these libraries with other team members or on different PCs. Make sure this directory is not write-protected.

**Custom libraries**

*Path for storage of custom libraries:* Custom libraries are containers for custom operators created by the user himself. For creating custom operators, a VisualApplets **Expert** license or the **VisualApplets 4** license is required. In this field, you can specify the directory where the custom operators are to be saved to and loaded from. Change this directory to another directory (e.g., to one on a network

drive) when you are sharing the custom operators with other team members or on different PCs. Make sure this directory is not write-protected.

**Script collection**

*Path for storage of your Tcl script collection:* VisualApplets allows you to create Tcl scripts and to make use of the individual Tcl procedures as library elements. In this field, you can specify the directory where your Tcl scripts are to be saved to and loaded from. Change this directory to another directory (e.g., to one on a network drive) when you are sharing the your Tcl script collection with other team members or on different PCs. Make sure this directory is not write-protected.

**VisualApplets designs**

*Path for VisualApplets designs:* Here, you can define the default location where VisualApplets saves design files (*.VA) and loads them from. Make sure this directory is not write-protected, and clear the *Use previously selected directory* option.

*Use previously selected directory*: If the box is checked, VisualApplets suggests to save the *.va file into the directory you used last time for this purpose. If unchecked, VisualApplets suggests the directory specified in the field *Path for VisualApplets designs* for saving/loading design files.

**SDK projects**

Path for storing SDK projects: Select here the default location of the automatically generated source code („SDK project“) that demonstrates the usage and parametrization of the generated applet with the Framegrabber API. The SDK project is generated after the build of the hardware applet or on demand when you choose from the menu **Build** -> **Generate SDK Example**.

**Help**

Path to help: Points to the base directory where the files for operator and context help are installed. You can change this path to a central location in order to share the help files among different installations. Make sure the help files are available in the directory you specify.



**Where to store the hardware applets (*.hap files created during the build process)**

The directory for storing your final hardware applets (*.hap files) you define under 'System Settings' in the category *Global Build Settings* and here under *Generation of Hardware Applets / Path for storage of hardware applets (*.hap)*.

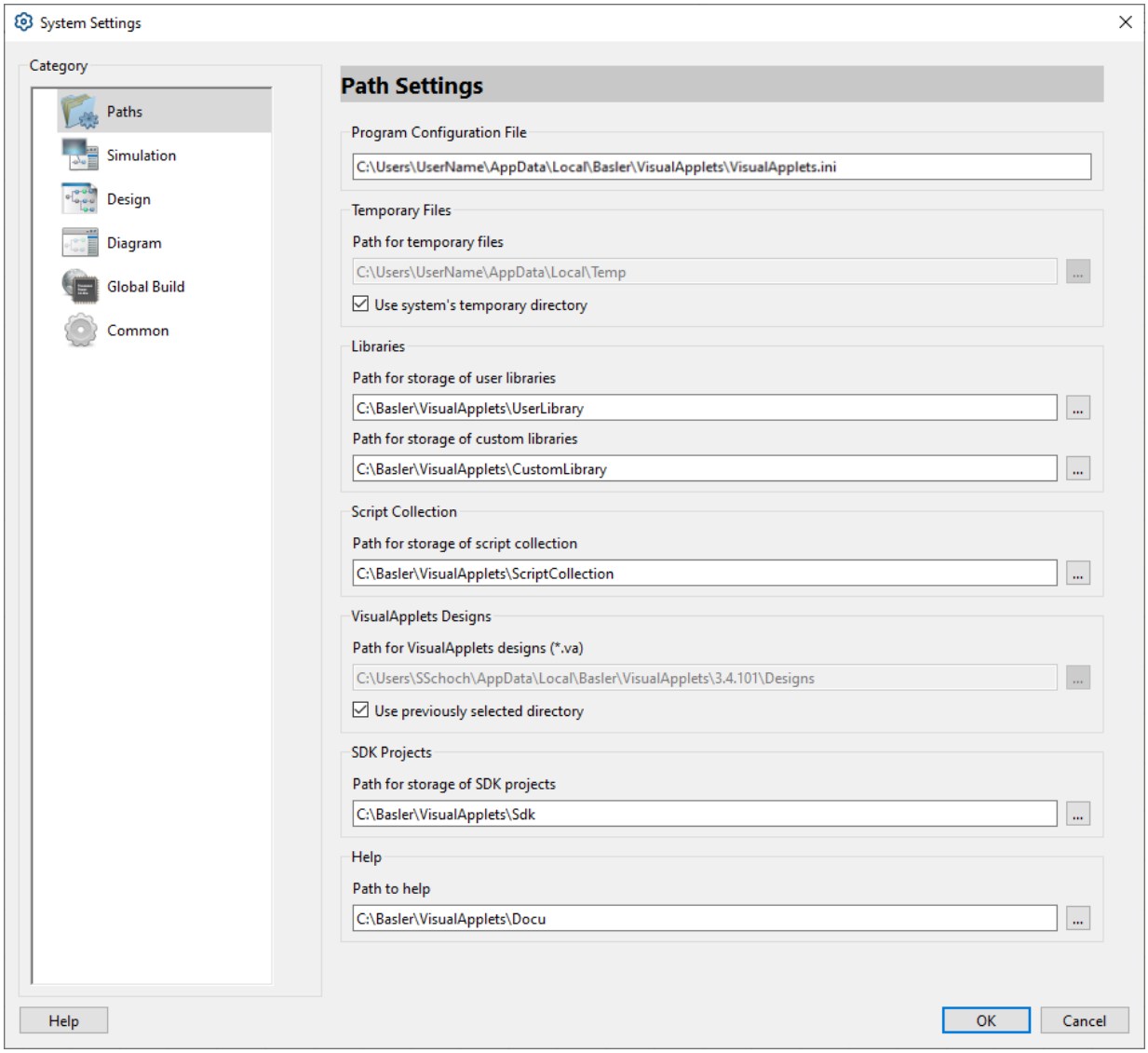



Figure 4.28. Dialog window for *Path Settings*

## 4.7.2. Simulation Settings

These settings affect the default behaviour of the simulation functionality of VisualApplets.

**Simulation Sources**

Location of images for simulation: Specify here the directory where VisualApplets should get the images for simulation. VisualApplets comes along with some sample images for demonstration. But in a real world, you will have your own images, similar to those which have to be processed. Specify the directory where your test images are.

Use previously selected directory: If the box is checked, VisualApplets first suggests the directory where you selected an image for simulation from last time. If unchecked, VisualApplets always suggests the directory specified above for loading images.

**Simulation Probes**

Path for destination images: Storage location of image files which are created at simulation probes while the simulation is running. This setting comes up as suggested path when the user is saving the collected simulation results of a certain simulation probe.

**Temporary Image Files**

Path for temporary image files: Storage location of temporary files which are created while the simulation is running by modules processing simulation data. The specification of this location allows VisualApplets to pick up these files automatically, e.g. at a certain module, and to put them as a source to another part of the design or to another module within the same design.

Use VisualApplets' temporary directory: If you clear this option, make sure the directory you specify is not write-protected. If write-protected, VisualApplets cannot simulate data at all.

The results of the simulation are saved in a binary file format. Two different formats are available:

- Save Raw Data as *.rsd : Files are saved in the „RSD" file format (image data and descriptive data).
- Save Raw Data as *.raw: Files are saved in the „Raw" file format (image data only).

Normally, the user doesn't need to edit any of these settings.

**Simulation Options**

VisualApplets (version 3.2 and higher) is equipped with a new simulation Engine. The new engine gives a very realistic depiction of the data flow and allows simulating blocked image data streams. The new engine is enabled per default. You can disable the new engine here in cases where you prefer the old simulation engine.

**I/O Image Library**

Here, you have the choice between two libraries: You can define to use either the Image Magick libary for loading and saving images, or the native implementation of VisualApplets. The Image Magick library is selected per default.

Below, you can set the compression parameters for saving probe images.

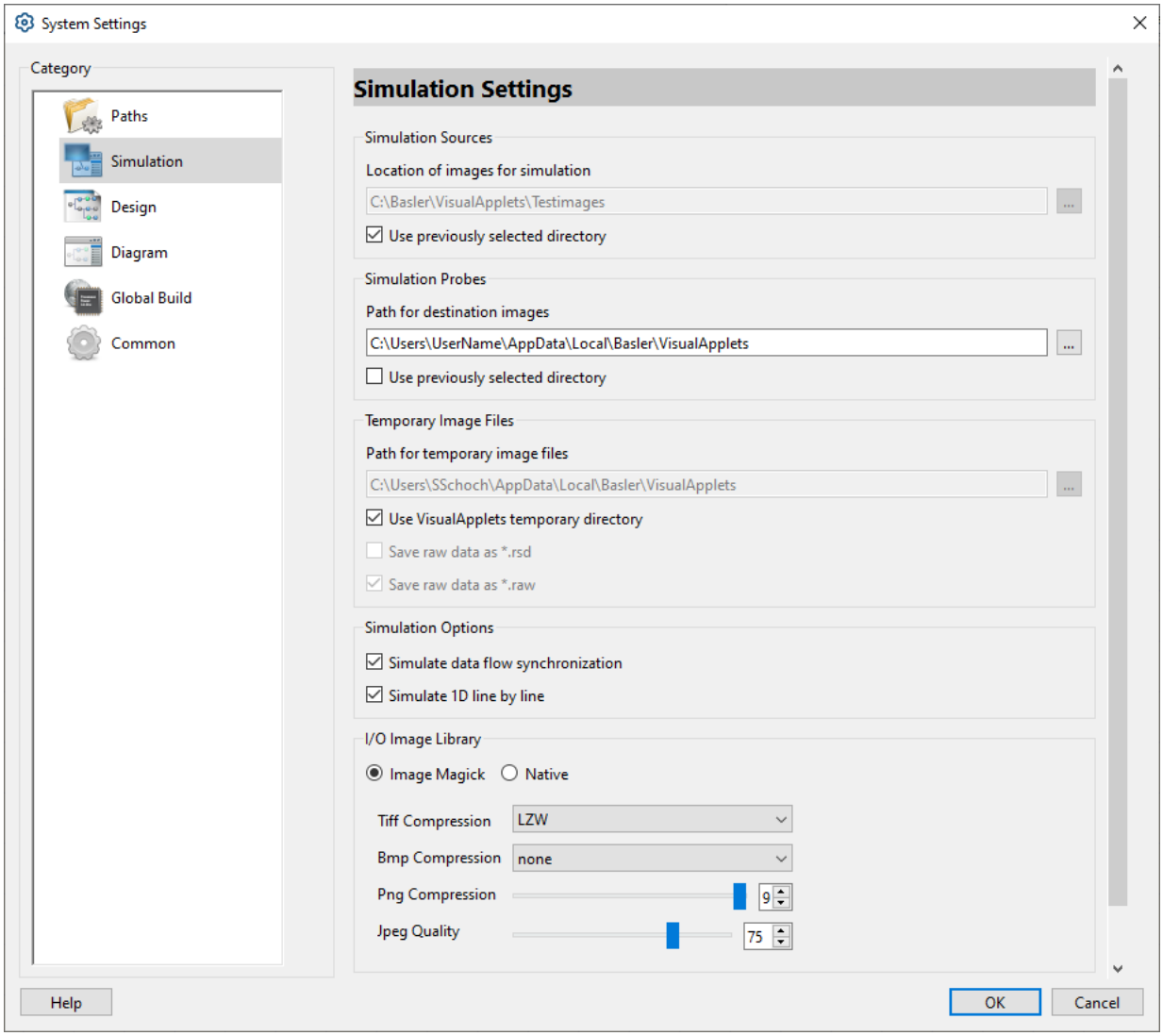



Figure 4.29. Dialog window for *Simulation Settings*

## 4.7.3. Settings for New Designs

These settings allow the user to define the default target platform the applets created in VisualApplets (target hardware applets) will run on. To set this settings is helpful when a developer's work is focused on certain frame grabbers („Hardware platform") or certain operating systems of the runtime („Operating system platform").

**Default platform for target hardware applets**

Hardware platform: The hardware platform selection shows all available hardware platforms (frame grabber types) which are currently available at the installation of VisualApplets. The platform you specify here will by suggested by the program as the default platform when you create a new design by using the menu item **File** -> **New**.

Example: If you create a project based on a microEnable IV – VD4-CL frame grabber, you can set this hardware platform here as the default platform. It will be suggested by the program when you create a new applet design.

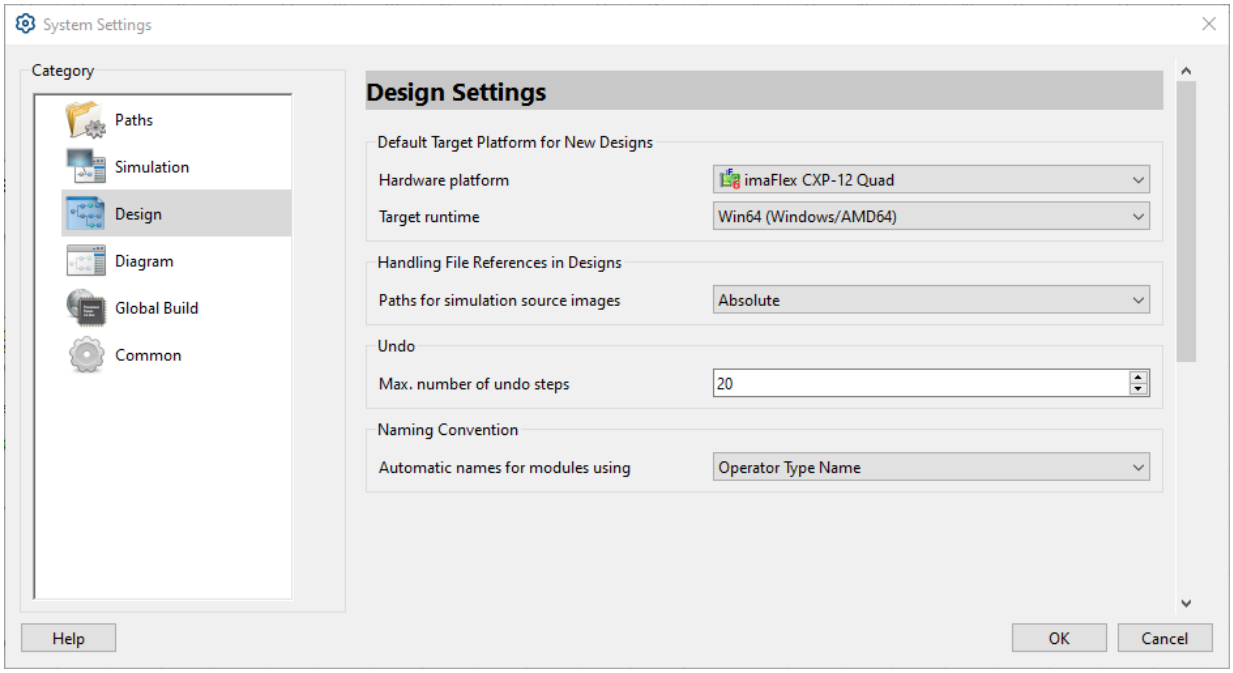



Figure 4.30. Example: If you always create applets for a Win64 system, you can set this operating system platform here as the default platform

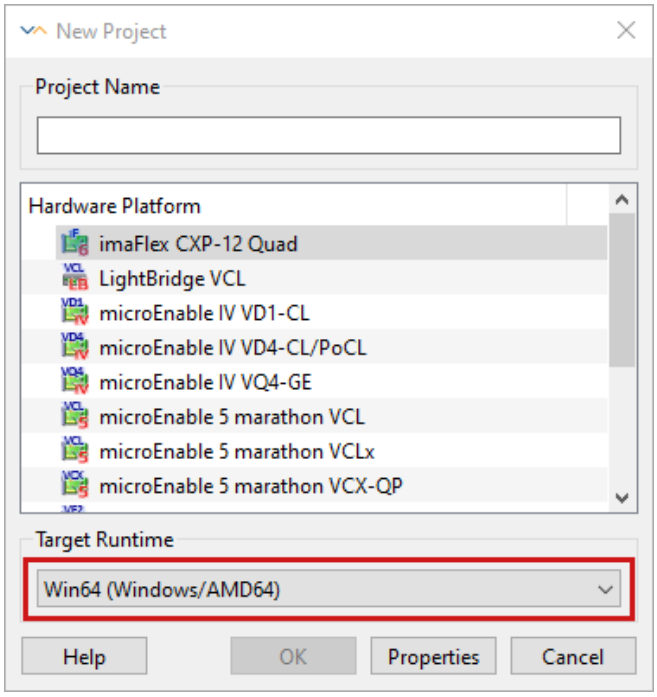



Figure 4.31. Example: Win64 will be suggested by the program when you create a new applet design

Operating system platform: The operating system platform selection shows all available operating systems for which applets can be created. The platform you specify here will by suggested by the program as the default platform when you create a new design by using the menu item **File** -> **New**.

**Tip**

Please note that the settings you enter here will come up as suggestions only. You can change both platforms any time, while creating a new applet, during the design process,

or even when you are already done with the applet design. The information about target hardware and target operating system is stored in the according *.VA design file.

### 4.7.4. Diagram Settings

This set of settings is related to the design window of VisualApplet.

**Extended Link Capture**

Defines the size of the capture area around the input or output ports of selected modules. Increasing the capture area allows an easier and faster creation of links for individual module ports since the user doesn’t need to hit the small ports exactly.

**Layout presettings for new designs**

Defines the default layout dimensions for new VisualApplets designs. This setting comes into effect when a new VisualApplets design is created via File/New. Please note that the layout dimensions of an existing design can be changed anytime via menu item Design/Diagram Layout Settings/ and are stored for each VisualApplets design individually.

**Grid**

Defines the default distances of grid lines in the design panel. This setting comes into effect when a new VisualApplets design is created via File/New. Please note that the grid line distance of an existing design can be changed anytime via menu item Design/Diagram Layout Settings/ and is stored for each VisualApplets design individually.

**Link Appearance**

These settings define the visualization of the connection links which are combining different modules at a design. Changes of these settings take effect immediately if you have a design file opened.

**Link Tool Tip**

Defines the information which is displayed at the tool tip when the user moves the mouse cursor over a link. Changes of these settings take effect immediately if you have a design file opened.

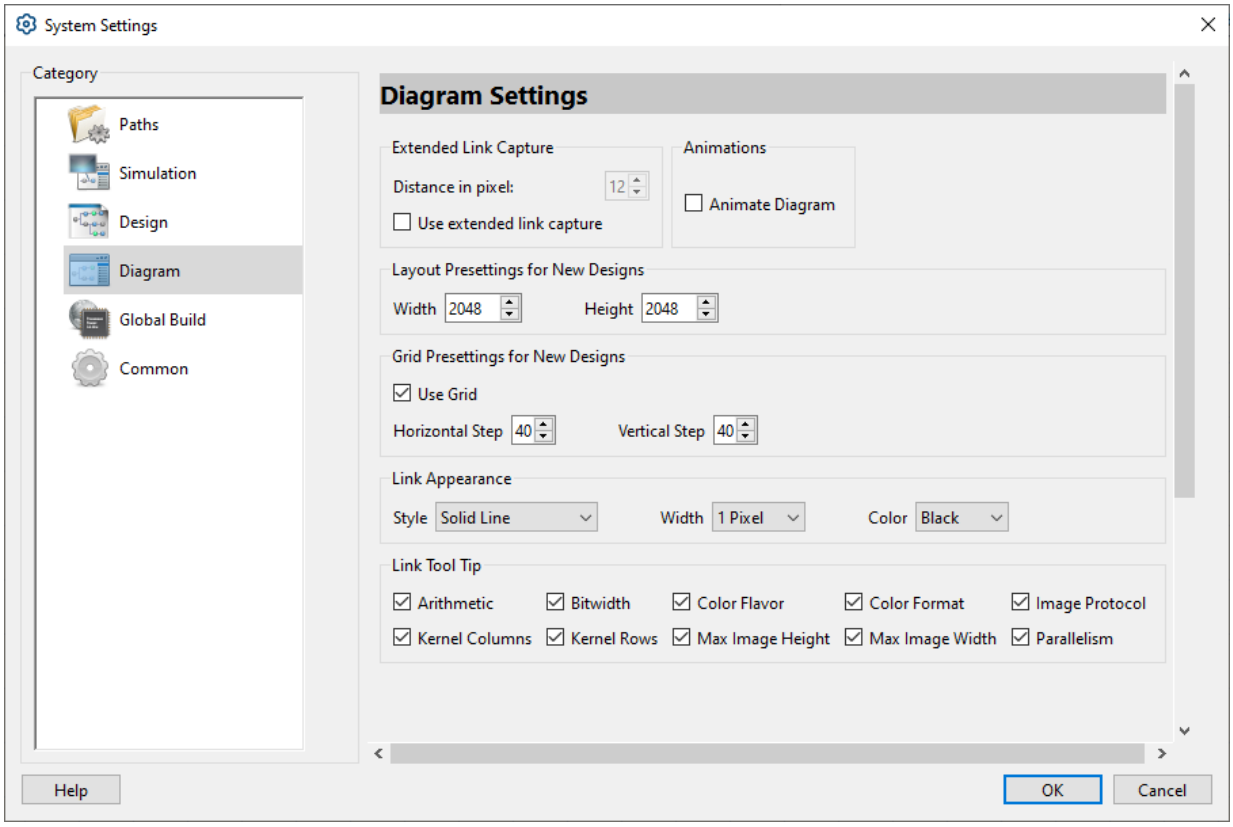



Figure 4.32. Dialog window for *Diagram Settings*

## 4.7.5. Global Build Settings

The settings of this category allow customizing the process of creating applets.

**Specific Synthesis Directory**

Here, you can specify the directory where temporary files created during the build process will be generated. These files will be deleted after the applet is generated. (See also **Build trace files** below)

**Generation of Hardware Applets**

Here, you can specify the location where generated hardware applets (*hap files) will be stored.

*Create subdirectories for hardware platforms:* Generates an additional subdirectory according to the platform name.

*Copy applets to runtime directories:* Copies the resulting files to the Framegrabber SDK installation folder (if installed) in order to load the files immediately to a present frame grabber.

**Build Trace Files**

*Keep build trace files:* Here, you can specify a directory where the intermediate files (which will be generated during the build process) can be copied into for detailed analysis of these files or for traceability reasons.

**Special Settings**

*Style of synthesized design netlists:* Select here the style for netlist generation:

- Use *Optimized* when building timing-critical designs to try to achieve a better timing closure in the build flow. VisualApplets will generate the FPGA netlist with an alternative naming scheme which

in some cases helps the Xilinx tool flow to generate a better distribution of logic on the FPGA. The setting of environment variable VA_NETLIST_STYLE has no influence.

- If set to *Default*, the environment variable VA_NETLIST_STYLE is consulted:
  - If the environment variable set to value 2, the alternative naming scheme is used for FPGA netlist generation. The alternative naming scheme in some cases helps the Xilinx tool flow to generate a better distribution of logic on the FPGA.
  - If the environment variable set to value 1, the legacy naming scheme (VisualApplets 3.1 and lower) will be used.
- If set to *Legacy*, the legacy naming scheme (VisualApplets 3.1 and lower) will be used. The setting of environment variable VA_NETLIST_STYLE has no influence.

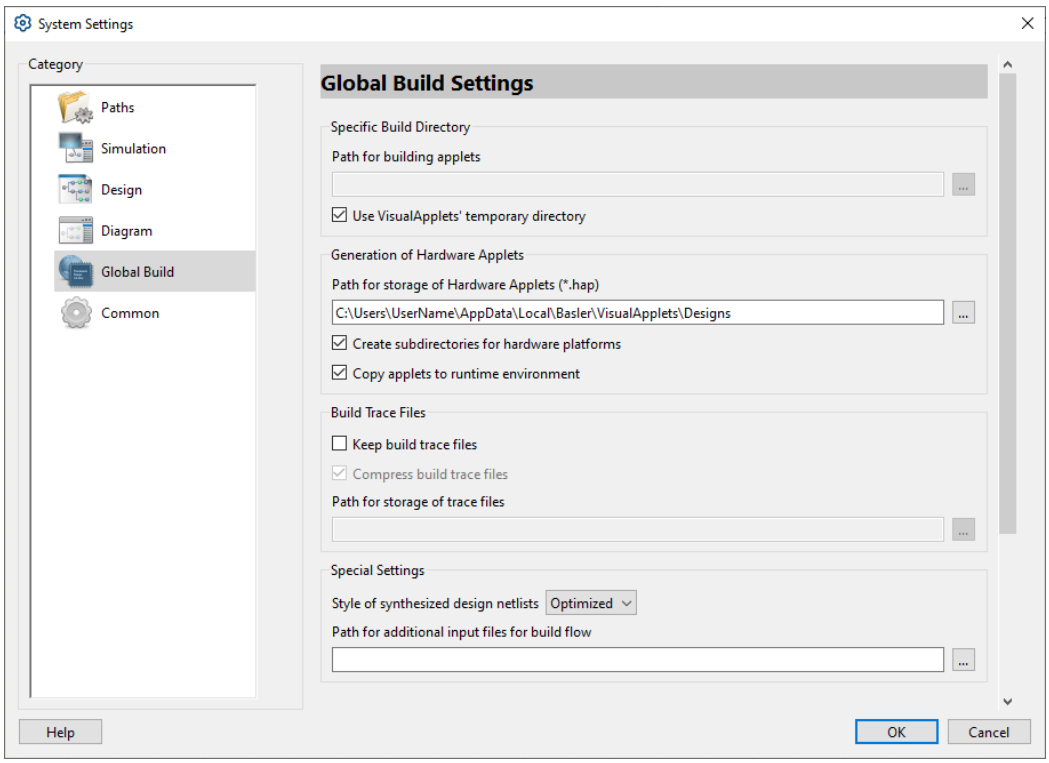



Figure 4.33. Dialog window for *Global Build Settings*

## 4.7.6. Common Settings

Here, you can select your preferred language for displaying the user interface of VisualApplets, define the toolbar icon style, define how many items are displayed in the recent design list and define the upsate bahvior of VisualApplets. .

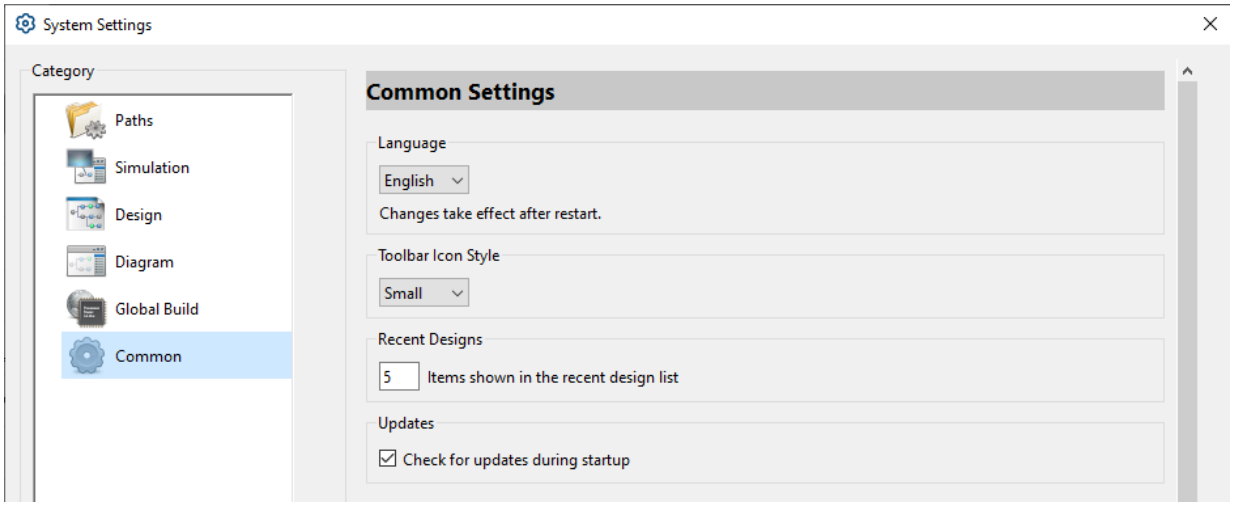



Figure 4.34. Dialog window for *common settings*

# 4.8. Design Settings

Besides the program settings it is possible to define default settings for each VisualApplets project. These setting will be stored in the `*.va` file and comprise the target runtime, project details as well as diagram layout settings.

## 4.8.1. Target Runtime

A VisualApplets hardware applet (HAP) can only be run on a single target runtime e.g. Windows 32Bit, Windows64Bit, Linux 32Bit, Linux 64Bit or QNX 32 Bit. When a VisualApplets project is build i.e. transformed into a HAP (See Section 3.12, 'Build') the HAP can only be used on the previously selected target runtime. The build step allows the selection of the target runtime. However, a default target runtime can be defined for each project individually.

To set the default target runtime of a project select **Design** -> **Change Target Runtime**. From the list, one of the supported target runtimes can be selected as shown in the next figure. This target runtime is now used for your project. Keep in mind to save your project file.

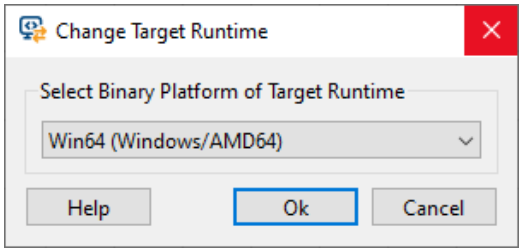



Figure 4.35. Target Runtime Project Setting

The default setting for new projects can be changed in the system settings. See Section 4.7, 'System Settings' for more information.

## 4.8.2. Project Properties

For each project, a project name, a version number (string) and a description can be added. To edit these attributes of your project, click **Design** -> **Properties** Enter any string values as shown in the next figure. Do not forget to save your design after modification. The entered values are shown in the project info dock window. After Build, this information is included in the applet and displayed in microDisplay as parameter values under "Applet Properties".

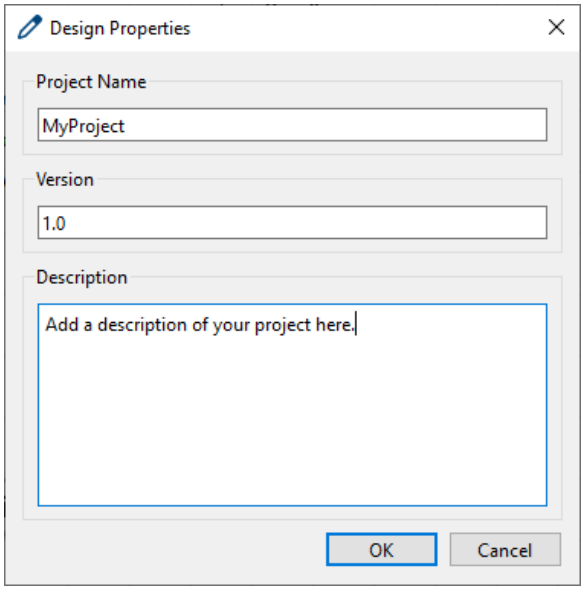



Figure 4.36. Editing the Design Properties

### 4.8.3. Diagram Layout Settings

To change the diagram layout settings of a project click on **Design** -> **Diagram Layout Settings** (see next figure). You can change the design area size of a project. By default, the size is 2048 by 2048 pixel. Moreover it is possible to enable or disable the grid and change its step size. The grid can also be enabled and disabled from the icon in the *View* toolbar. Do not forget to save your design after modification.

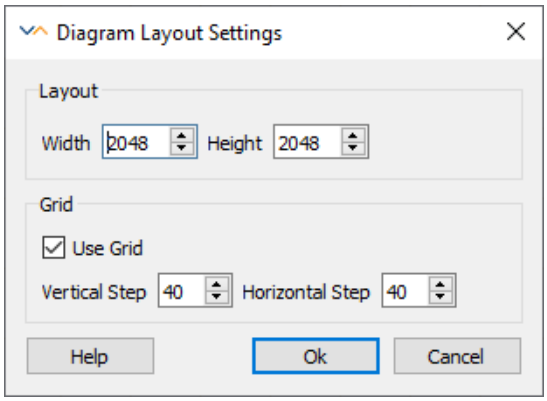



Figure 4.37. Diagram Layout Settings

## 4.9. Build Settings

To actually generate (build) the hardware applets (see Section 3.12, 'Build'), VisualApplets uses a whole tool chain that works in the background.

For each hardware platform, VisualApplets provides a default build setup that is optimized for the specific hardware platform.

Based on the hardware platform you selected for your design in the design settings, the according default setup for build is proposed by VisualApplets. This setup cannot be changed by the user, but can be replaced by a user-defined setup.



**You have to replace the default build setup by own settings if ...**

- You create designs for a microEnable 5 (ironman and marathon) or LightBridge platform.
- You work with Xilinx Vivado.
- You work with a Xilinx ISE version higher than 9.2.



**Recommended Xilinx Tools**

See Which Xilinx Toolchain and Version for Which Frame Grabber Platform [https://docs.baslerweb.com/visualapplets/installing-visualapplets#which-xilinx-toolchain-and-version-for-which-frame-grabber-platform] for detailed information about which Xilinx tool is required to build applets for which hardware platform.

**mE5 marathon and LightBridge:** Although applet build for mE5 marathon and LightBridge is possible with ISE 14.7, Basler recommends to use Xilinx Vivado since applet build (synthesis) with Vivado is much faster for these platforms.

### 4.9.1. Defining Build Settings

#### 4.9.1.1. Using one Parameter Set as Active Configuration

Each time the program performs the build steps, a certain set of build parameters will be used.

There are several sets of build parameters that come with VisualApplets. You can add additional sets.

For building an applet, one of these sets has to be specified as the “Active configuration”. This configuration will be used and the tool chain will be controlled accordingly.

All setting information is stored in the file “SynthesizeSettings.xml” at your local installation.



**Activated and Current Parameter Set**

To open the *Build Settings* dialog, click **Settings** -> **Build Settings**.

The Dialog opens. The activated parameter set displayed:

- The activated parameter set is indicated by a green tick in the left-hand panel of the Build Settings dialog.
- The activated parameter set is used for build.
- The activated parameter set stays activated even if you select another parameter set to be displayed. (To display a parameter set, click on its name in the left-hand panel

of the Build Settings dialog. The selected parameter set will be displayed in the right-hand panel of the Build Settings dialog.)

- The parameter set displayed you can edit, copy, save, or delete (see below).

In the following, we will refer to the parameter set displayed as the "current parameter set".

### 4.9.1.2. Defining New Parameter Sets

The set of parameters itself consists of parameters for starting the external applications (Xilinx tools) that perform certain steps of the workflow, and of parameters for controlling these tools. You can define a certain set to be the default configuration (for starting a build of the current design) by checking “Active configuration”, and you can add a comment to describe the purpose of this specific set.

1. To open the *Build Settings* dialog, click menu **Settings** -> **Build Settings**.

   The Dialog opens. The activated parameter set is displayed.

   **Before Adding the First Alternative Parameter Set**

   As long as no additional parameter set for the target hardware of your design has been created, the default build settings are displayed.

   You cannot make any changes to these default settings, therefore, all input fields are deactivated.

   You also cannot uncheck “Active configuration” as there is no alternative to the default yet.

2. Click **Add** to create a new set of parameters.

   You will be asked for hardware platform information. You can specify here which parameter set you want to load as a basis for creating your own parameter set. In most cases, it is quite sensible to select here the target hardware of your design.

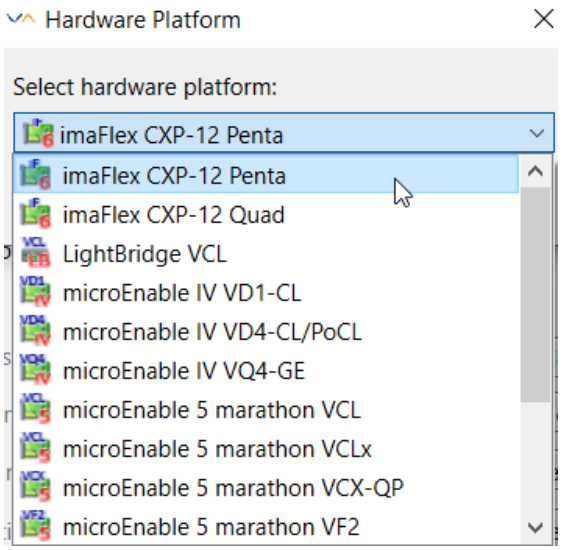



Figure 4.38. Selection of Hardware Platform

3. Click **OK**.

   Now, you can edit the first fields.

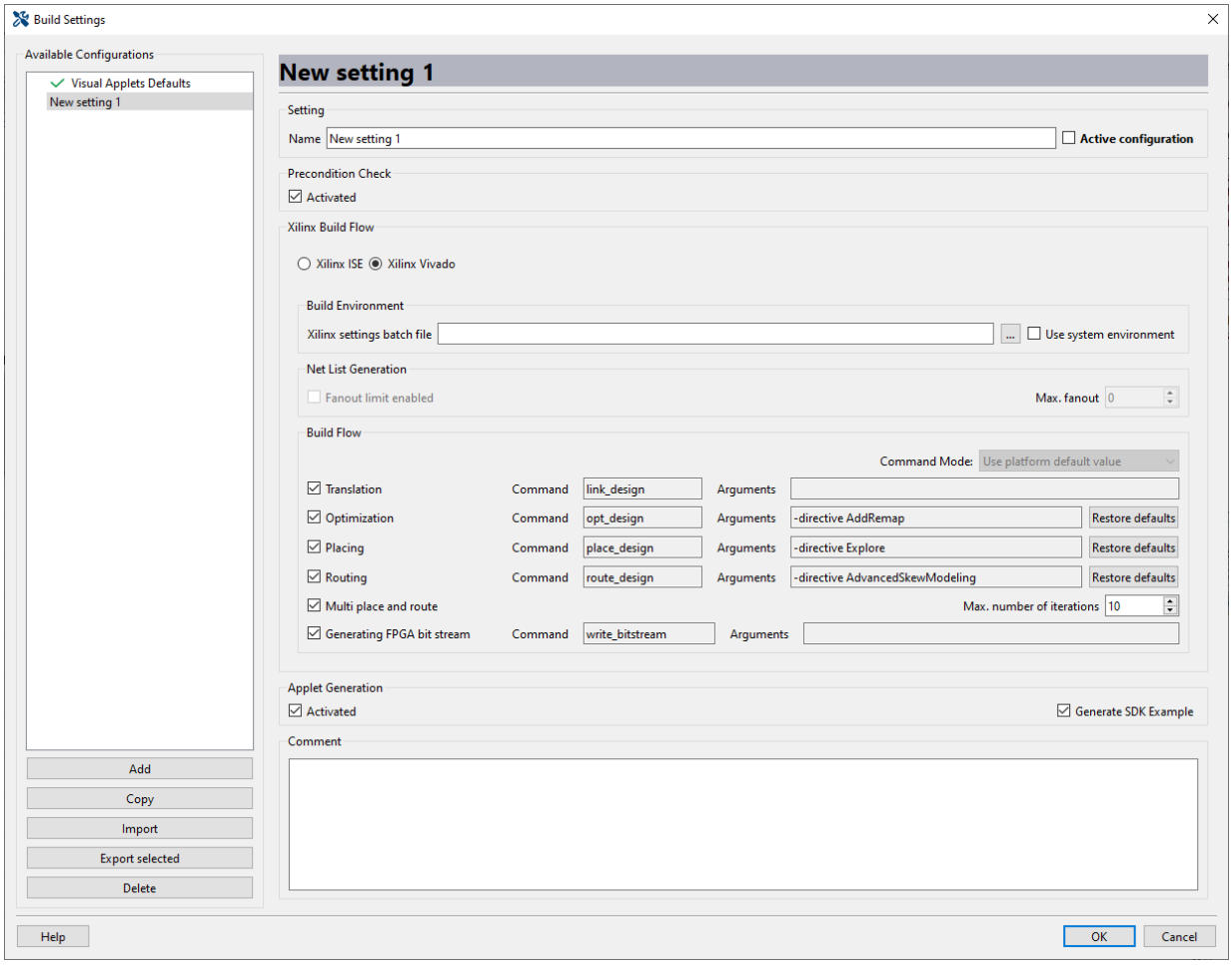



Figure 4.39. Build Settings Window

4. **Name:** Give a name to your new set of properties for identification. This name will show up in the configuration dialog and in the build protocols.

5. **Active configuration:** If you want to use this set of parameters when building your applet, check the box *Active Configuration*.

> **Active Configuration**
>
> The set of parameters you define as **Active configuration** will be used as often as you execute a build.
>
> When you change the hardware platform you selected as target hardware platform of the design: Define new build settings for this new platform.

6. **Precondition Check:** In default mode, the Precondition Check is activated. The Precondition Check verifies that the FPGA type on the target hardware of your design is supported by your Xilinx tool chain. In addition, in all designs for mE5 or LightBridge platforms: When this check box is activated, the information in operator *AppletProperties* is updated at each build, namely the fields *BuildTime* and *AppletUid*. Therefore, after a build has been completed, the design is in "unsaved" mode, since the parameter values in the operator *AppletProperties* have been changed during build.

> **Avoiding the Update of AppletProperties**
>
> Under specific circumstances you might not want to update the *AppletProperties* operator by a build. In this case, simply de-activate the check box *Preconditions Check*.

7. **Xilinx ISE/Xilinx Vivado:** Select the Xilinx tool chain you are going to use. Vivado is supported by all marathon and LightBridge platforms. If Vivado is not supported by the target hardware of the open design, this is stated directly in the GUI:

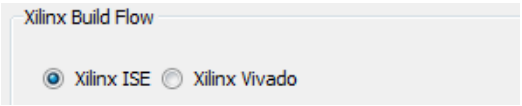



Figure 4.40. Vivado Supported by Target Hardware Design

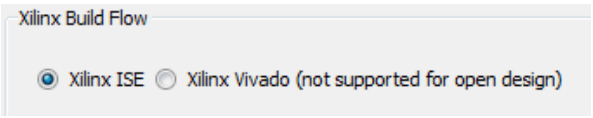



Figure 4.41. Vivado not Supported by Target Hardware Design

8. **Build Environment:**

   - When using a newer Xilinx tool (i.e., when developing with Xilinx Vivado or with Xilinx ISE in a version higher than 9.2):

     - De-activate the "Use system environment" checkbox, and

     - from your file system, select the Xilinx settings batch file that sets the environment to launch the external tools.

     Please follow the Xilinx documentation to make the best choice.

     You find the batch file in the Xilinx installation folder:

     - **ISE:** \Xilinx\14.7\ISE_DS\settings64.bat.

     - **Vivado:** \Xilinx\Vivado\2020.2\settings64.bat.

     Basler recommends to use the 64-bit Windows operating system when developing applets for microEnable 5 platforms. Make sure you select the batch file that matches the operating system you are using, e.g., "settings64.bat" which is the file for the 64-bit Windows OP.

   - If you use an older Xilinx ISE version (version 9.2 or older) **and** all environment variables are set at the operating system, activate the "Use system environment" checkbox.

9. **Build Flow:** Activate all options under *Build Flow*. Normally, all steps should be activated.

   Basler recommends to set the system under *Multi place and route* to 20 iterations.

   Keep the *Command Mode* at *Use platform default value*.

10. **Applet Generation:** Activate *Applet Generation*.

11. **Comment:** Enter a comment that describes the set of parameters you just created. The entries you make here have no effect on the actual build process.

12. Click the **OK** button.

Example: When developing for microEnable 5 or LightBridge, the following parameter values need to be set:

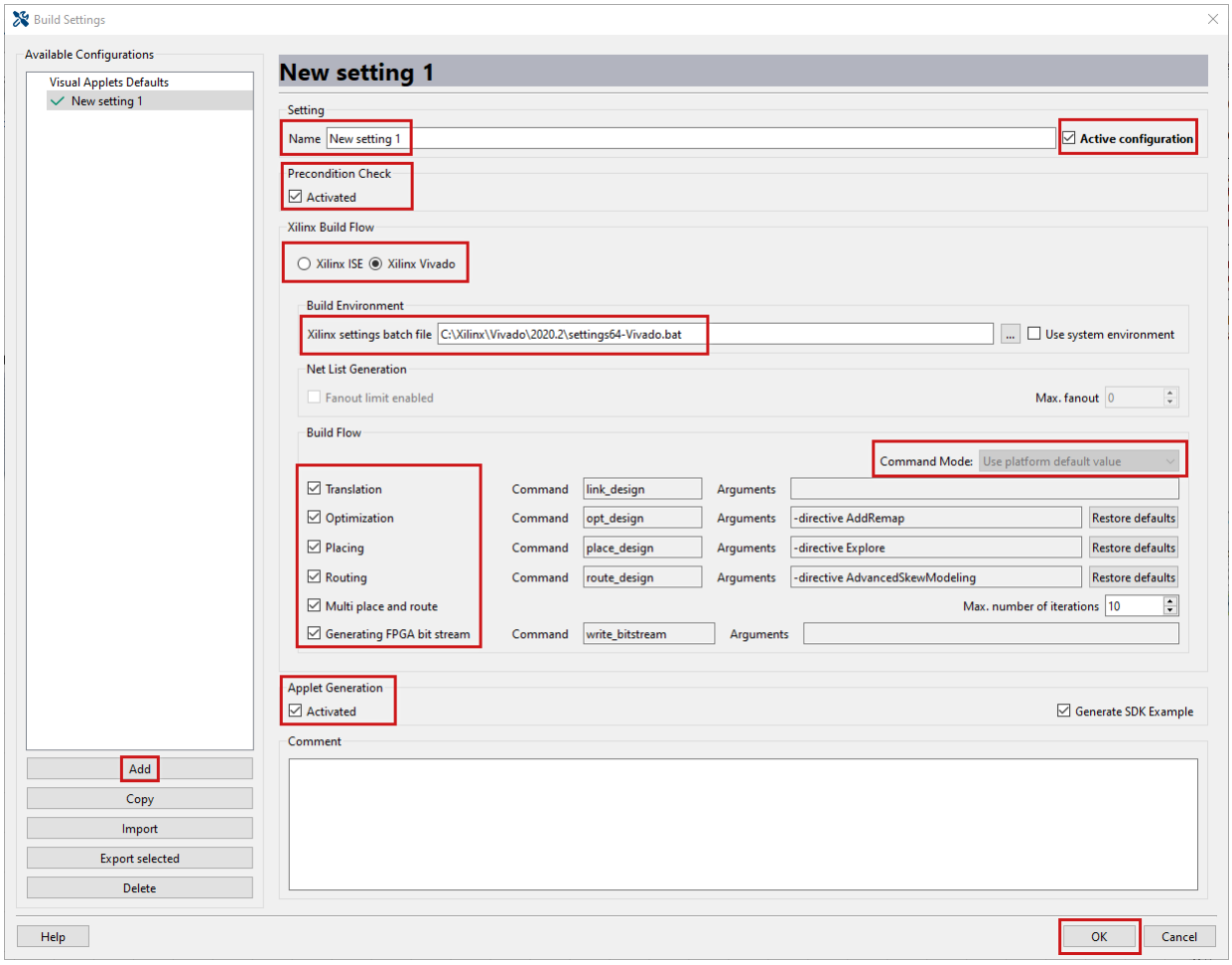



Figure 4.42. Parameter Set Example: Developing for microEnable 5 or LightBridge

### Disabling Build Flow Steps

Per default, all build flow steps (translation, mapping, ...) are activated:

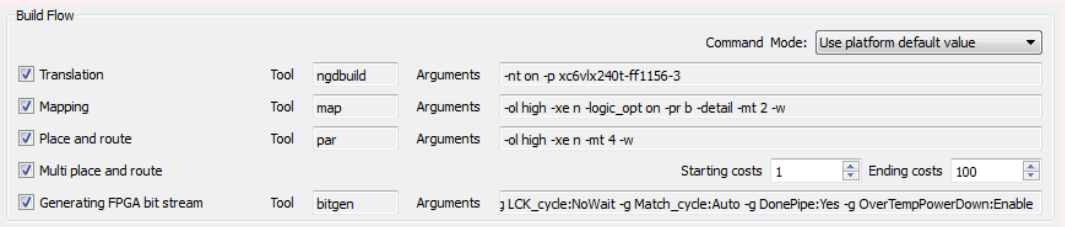



Figure 4.43. Defaut: All Build Flow Steps Activated

With the **Expert** license or the **VisualApplets 4** license you can de-select individual steps. However, the build steps depend on each other. They are listed in the order of their dependence:

- Translation is the first step that can always be executed.
- Mapping can only be executed if the step Translation has been executed before.
- Place and route can only be executed if the step Mapping has been executed before, and so on.

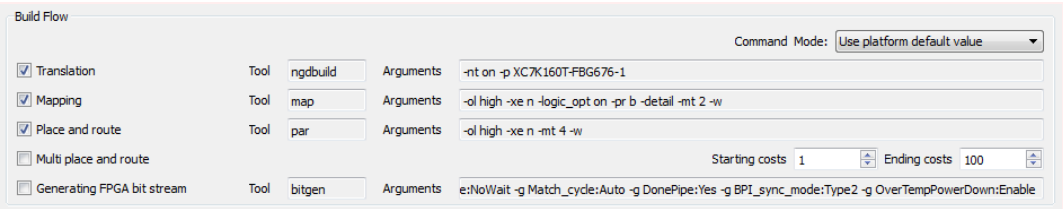



Figure 4.44. Subsequent Build Steps Deactivated

However, a previous step doesn't need to be executed within the same build run. It might as well have been executed in an earlier build run. As long as the files (trace files) that have been generated during an earlier build run are still available, it is possible to deactivate steps that come earlier in the "Build Flow" list and to activate steps that come later in the list.

To keep trace files of earlier build runs: Go to menu **Settings** -> **System Settings** -> Global Build. Under *Build Trace Files*, activate *Keep build trace files*. Deactivate *Compress build trace files*.

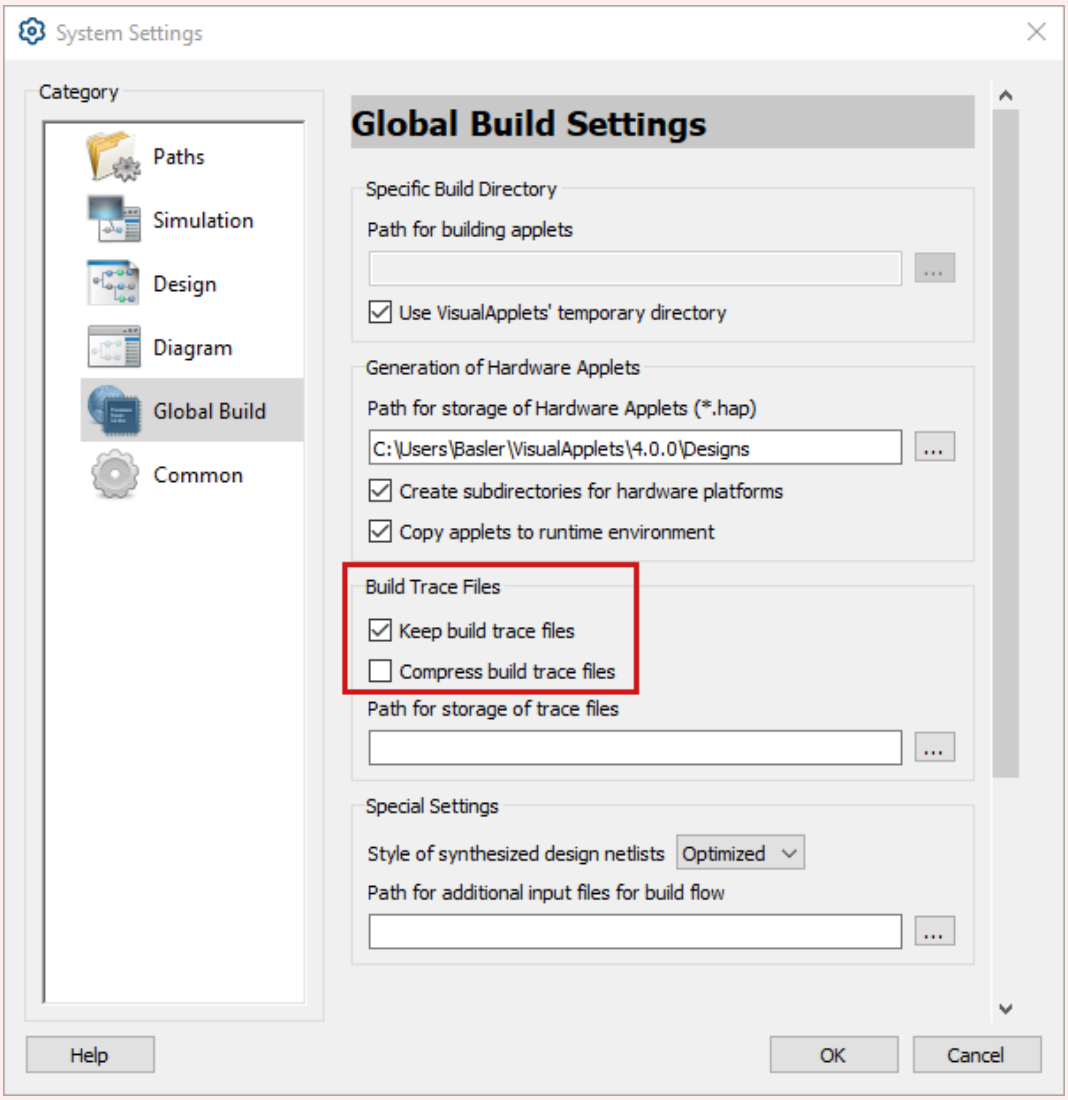



Figure 4.45. Keeping Build Files of the Individual Build Steps

If you keep the build trace files, and you have carried out all previous build flow steps once before, you can also activate only subsequent steps, e.g.:

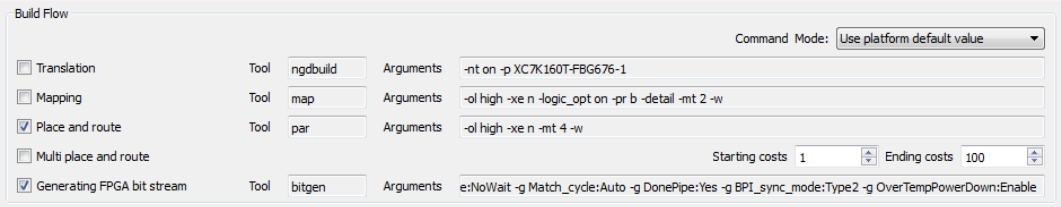



Figure 4.46. Keeping Build Files of the Individual Build Steps



## Command Mode Options

With the **Expert** license or the **VisualApplets 4** license you can change the **Command Mode**.

To change the **Command Mode**, go to *Command Mode* at the right upper corner of the *Build Flow* area.

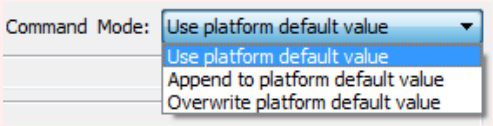



Figure 4.47. Command Mode Options

In the list, you have three different options:

- Use platform default value (Default)
- Append to platform default value
- Overwrite platform default value

---

**Use platform default value** (Default):

If you keep or select this option, VisualApplets displays the default values recommended for the hardware platform you specified as target platform for the open design.

VisualApplets also uses these default values when running the Xilinx tools (i.e., when the actual applet is build (synthesized) out of the open design).

Since default sets cannot be modified, it is not possible to edit the *Arguments* fields in the *Use platform default value* mode.

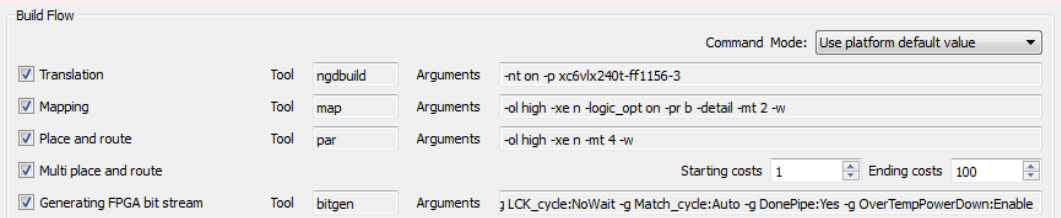



Figure 4.48. Command Mode "Use platform default value"

**Append to platform default value**:

If you select this option, you can add information for the *Mapping* and *Place and route* tools to the basic arguments provided by VisualApplets.

During build, VisualApplets adds the parameters you defined in the *Arguments* fields to the platform default values when calling the tool. **This setting is for experienced users only.**

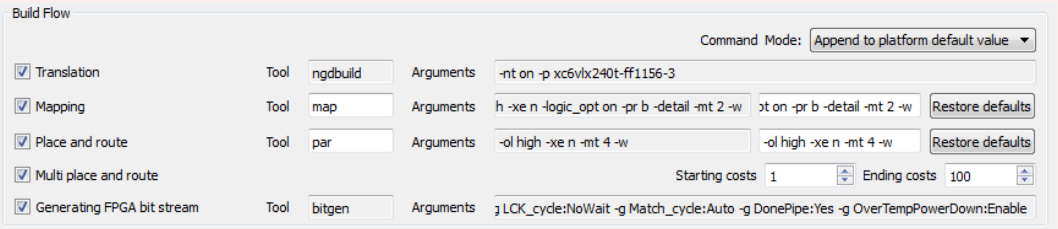



Figure 4.49. Command Mode "Append to platform default value"

**Overwrite platform default value**:

If you select this option, VisualApplets displays the default arguments for *Mapping* and *Place and route* for the target hardware platform you selected directly after creating the new applet set. You can define (overwrite) the *Mapping* and *Place and route* arguments stated here. VisualApplets now uses only the parameters you specify in these *Arguments* fields when calling the tools. **This setting is for experienced users only.**

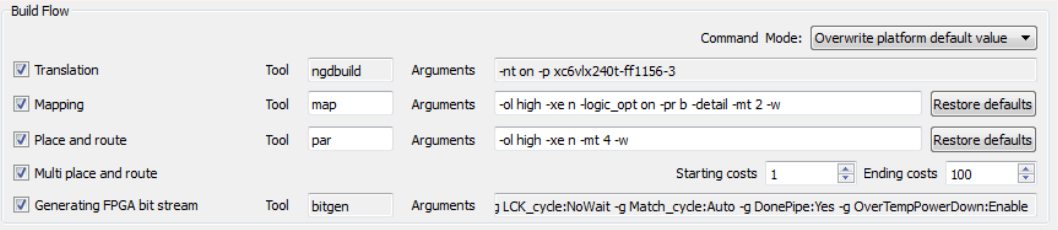



Figure 4.50. Command Mode "Overwrite platform default value"

### 4.9.1.3. Re-Using Parameter Sets

In the *Build Settings* window, you have some options for handling parameter sets:

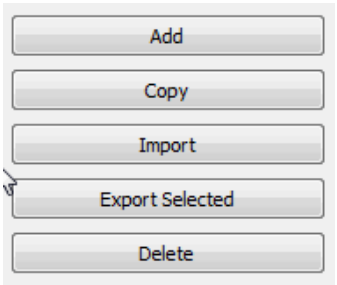



Figure 4.51. Handling Options

Clicking on

- **Add** creates a new parameter set which you can modify as you want (based on the hardware platform you select).

- **Copy** copies the parameters of the current set, creates a new set and pastes the parameters of the copied set into the new set.
- **Import** allows to import a certain set of parameters into the current installation of VisualApplets.
- **Export selected** saves the current parameter set to a file. The file can be used for transporting the data to a different PC or installation or for backup reasons. A dialog is displayed where you can specify a file name and location.
- **Delete** deletes the current set of parameters.

### 4.9.1.4. Experienced users only: Creating *.hap Files for Different Operating Systems Using the Same Build (mE 5 ironman, mE5 marathon and LightBridge)

For mE 5 ironman, mE5 marathon and LightBridge, the checkbox **Precondition Check** offers a specific function:

- If **Precondition Check** is activated, at each start of synthesis (build) the entries for BuildTime and AppletUid are updated. If you want to start a complete build (from netlist generation through to applet generation) you need to activate **Precondition Check**. Otherwise, later on (when using the applet on the frame grabber) the loaded applet cannot be reliably validated by the runtime environment.
- Specific option: If you have already completely built an applet, but decide you need another *.hap file for another target operating system: You can build a new OS-specific hap file out of the complete build to make your design usable on other target operating systems (i.e., for another target runtime). Proceed as follows:

  1. In menu **Settings**, select **System Settings**.
  2. In the window that opens, select category **Global Build**.
  3. Here, under **Build Trace Files**, activate the option **Keep build trace files** and specify a path.
  4. Go to menu **Settings**, menu item **Build Settings**.
  5. **Deactivate** option **Precondition Check**.
  6. Deactivate all other options under **Xilinx Build Flow**.
  7. Activate **Applet Generation**.
  8. Under Defaults, activate **Active Configuration**.
  9. Start the build of the OS specific *.hap file.

  Especially with complex designs, this option allows you to save time.

## 4.10. Tcl Scripting

Tcl provides the power of a programming language while being easy-to-learn. Using it can be as simple as calling commands in a shell. At the same time advanced programming techniques can be used to set up sophisticated processing schemes.

To combine VisualApplets with the power of a scripting language allows for an enormous increase of productivity.

You can

- **Automatize the creation of designs:**

  Use scripts to assemble components of your design controlled by parameters. Set up libraries of processing pipelines and create bunches of different designs automatically.

- **Automatize the simulation of designs:**

  Tcl allows you to define very extensive simulation runs: You can, for example, specify series of simulation runs over thousands of images stored somewhere in your file system. Or, you can define various simulation runs where different sets of design parameters are used for simulation.



**Documentation**

You find information on how to use the VisualApplets Tcl console and an extensive command reference at TCL Documentation [https://docs.baslerweb.com/visualapplets/overview-of-tcl-scripting].



**Availability**

The VisualApplets TCL scripting feature is part of **VisualApplets Expert**.

To use TCL scripting, you need to hold either an **Expert** license or the **VisualApplets 4** license.

## 4.11. Script Collection (Tcl)

You can make the individual procedures in your Tcl scripts available directly in the VisualApplets GUI.

VisualApplets allows you to define a script collection (library) containing sorted commands that each call a specific procedure in a specified Tcl script. The commands are graphically available directly on the VisualApplets GUI. As soon as you start a command via mouse click or per Drag and Drop, the underlying procedure is carried out. You can monitor what happens in the Tcl console of the VisualApplets program window.

You can define a script collection either in a graphical way, using the VisualApplets GUI, or creating/editing an XML file that defines the script collection.

As all information regarding your script collection is stored in XML files, you can use version control systems not only for the individual Tcl scripts, but also for the structure and content definition of your script collection.

The commands are available in pane Script Collection:

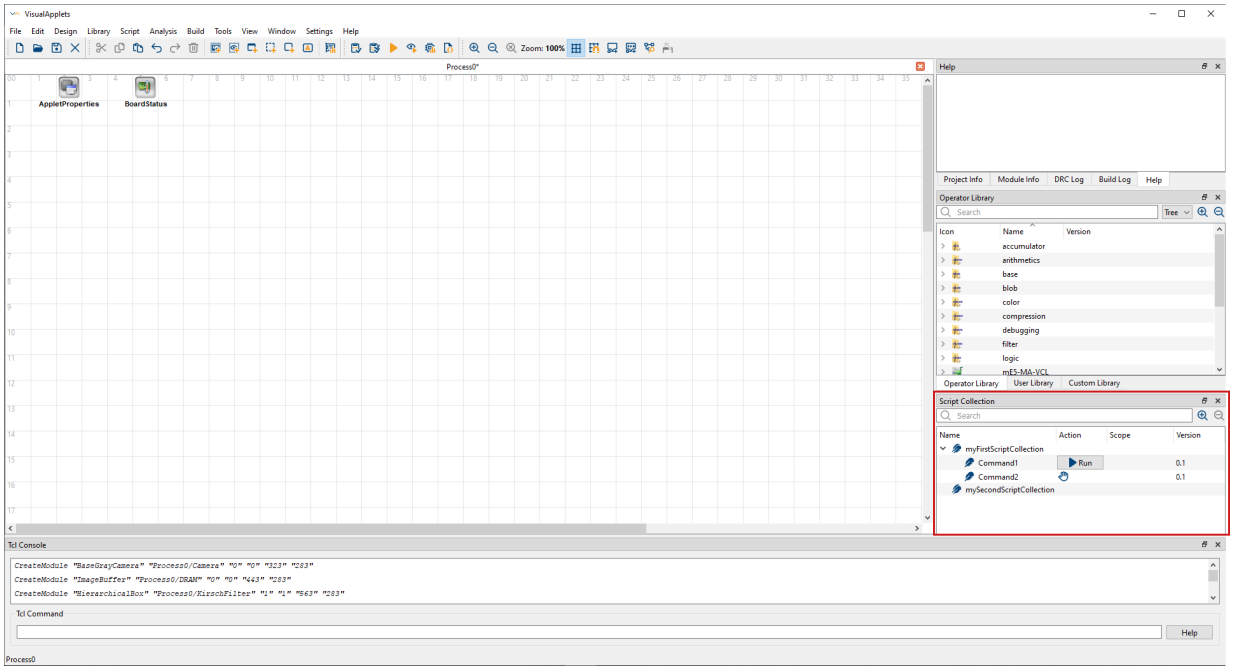



Figure 4.52. Script Collection in the VisualApplets program window

**Documentation**

You find information on how to use the VisualApplets Tcl console and an extensive command reference at Creating Script Collections (Script Libraries) [https://docs.baslerweb.com/visualapplets/creating-script-collections] and at Command Reference [https://docs.baslerweb.com/visualapplets/tcl-command-reference].

**Availability**

The VisualApplets TCL scripting feature is part of **VisualApplets Expert**.

To use TCL scripting, you need to hold either an **Expert** license or the **VisualApplets 4** license.

## 4.12. Tcl Export

VisualApplets allows you to export VisualApplets designs as Tcl script (*.tcl).

You can use the script format for the following purposes:

- revision control
- comparing versions automatically (by creating "diffs")

However, some information is lost during export to Tcl script. Therefore, you should always save your design in `*.va` format. The Tcl file is not qualified to be used as the primary data format.



**Save Design in *.va Format**

Use the `*.va` file format as primary data format for your VisualApplets designs.

The Tcl script does not contain the full design information.

### 4.12.1. Exporting to Tcl

The following information is lost during export to Tcl script:

- The local modifications you made in a user library element instance (to adapt the instance to the surrounding design) are lost. If you re-import a Tcl script into VisualApplets, the instance is inserted as a fresh copy of the user library element. The content of the instance in your design has not been saved in Tcl.
- The content of info boxes is kept in Tcl format. However, the formatting of info boxes is lost.

You should always save your design in `*.va` format, too. The Tcl file is not qualified to be used as the primary data format.

To export your design to a Tcl file (`*.tcl`):

1. From the menu, select **File** -> **Export** -> Design to Script....

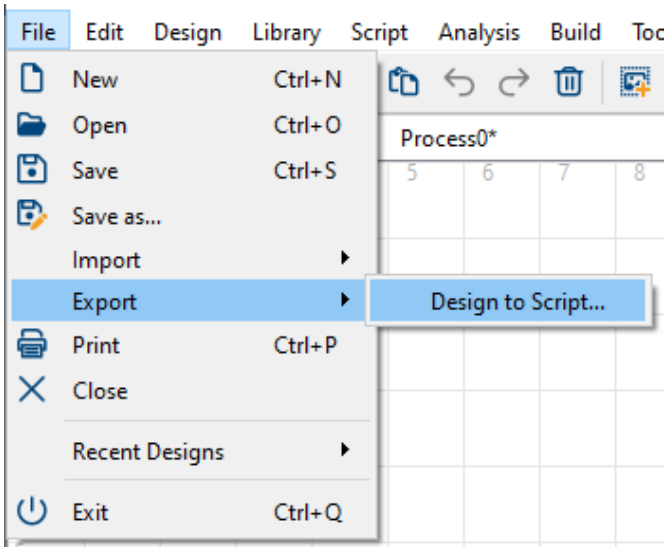



Figure 4.53. Exporting a Design

2. Select a location for the `*.tcl` file containing your design, and click on **Save**.

Now you can check in the Tcl file into your version control system, or find differences between different versions of the design using a "diff" tool.

### 4.12.2. Importing from Tcl

To import your `*.tcl` design file into VisualApplets:

1. From the menu, select **File** -> **Import** -> Design from Script....

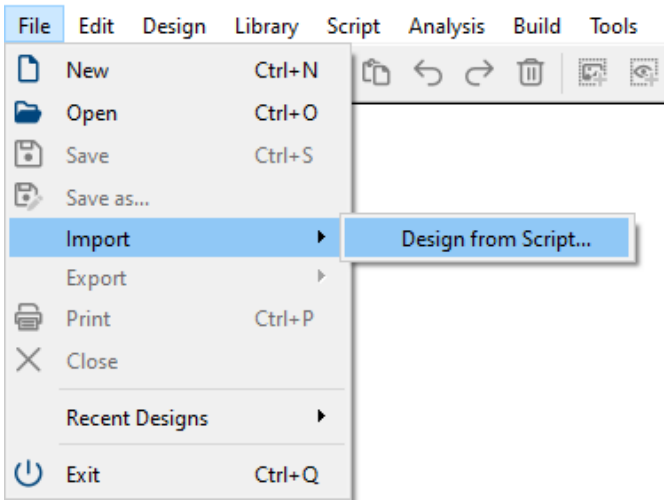



Figure 4.54. Importing a Design

2. Browse to the location of the `*.tcl` file containing your design, and select the ***.tcl** file.
3. Click on **Open**.

Now your design is opened in VisualApplets, and you can

- continue editing your design within VisualApplets,
- save your design file as `*.va` file.

**Always use *.va File Format as Primary Data Format**

Always use the `*.va` file format as primary data format for your VisualApplets designs.

Never use the Tcl script format as primary data format, as the Tcl script does not contain the full design information.

### 4.12.3. Tcl Import/Export Command Reference

In the generated Tcl source file the following VisualApplets-specific commands are used.

**Syntax**

**Optional parameters:** The notation of optional parameters follows the Wirth Syntax Notation (brackets).

**Parameter FullModuleName**: Hierarchical path starting at process level. The hierarchies are separated by a slash.

| Command | Parameters | Description |
|---|---|---|
| **Creating, Loading, and Saving Designs** | | |
| CreateDesign | Name<br><br>Platform | Command for creating a new design with the name "Name" for a hardware platform |

| Command | Parameters | Description |
|---|---|---|
| | | "Platform". For specifying the target platform of the design, you can either use<br>• the VisualApplets library name for the platform (e.g., "mE4VQ4-GE"), or<br>• the full name of the platform (e.g., "microEnable IV VQ4-GE").<br>Using capitals in the names has no effect. |
| SetDesignProperty | Key Value | Use this command to enter design properties. There are tree values you can use as key:<br>• ProjectName (the name you want the project to give)<br>• Version (version of the design)<br>• >Description (description of the project). |
| SetDesignClock | NewFrequency | Command for setting the basic clock frequency of the FPGA (in MHz). |
| **Setting up a Design** | | |
| CreateProcess | - | Creates a new process |
| CreateModule | OperatorType<br>FullModuleName<br>[ InPortMultiplicity<br>[ OutPortMultiplicity<br>{NamedPortMultiplicity}<br>] ]<br>xPos<br>yPos | Creates an instance (Module) of an operator (OperatorType).<br>The position of the module in the design window you define via parameters xPos and yPos.<br>With FullModuleName you define the module's hierarchical position within the design and it's name after instantiation (e.g., Process0/branch).<br>Optionally, you can use the value pair InPortMultiplicity and OutPortMultiplicity to specify the multiplicity of the vector inputs and vector outputs of the module.<br>If you use this value pair, you define the number of ports (in operators with variable port numbers, like BRANCH) for the module (which is a specific instance of an operator).<br>In addition, you can specify the multiplicity of individual vector ports (NamedPortMultiplicity). You can use NamedPortMultiplicity multiple times in one command line.<br>The format is PortGroup:Number.<br>Example: CreateModule BRANCH Process0/branch 0 0 O:3 100 100 |
| ConnectModules | FromModule<br>FromModulePort<br>ToModule<br>ToModulePort<br>[ x1 [y2 x3] ] | Command for connecting modules. For FromModule and ToModule, use hierarchical names (names containing the hierarchical position of a module within a design plus actual module name, e.g., (e.g., Process0/branch).<br>Nodes within hierarchical boxes are identified via the prefix "INBOUND#" or "OUTBOUND#". Example: "OUTBOUND#O001". |

| Command | Parameters | Description |
|---|---|---|
| | | The optional parameters x1, y2 and x3 you use to define the geometry of the connecting polygon (horizontal line from FromModuleOutputPort to x1, vertical line to y2, horizontal line to x3, vertical line up to the height of ToModulePort, horizontal line to ToModulePort). |
| CreateComment | FullModuleName<br>Text<br>xPos<br>yPos | The content of Text can contain Unicode characters. If you use characters that are not part of ASCII, use the according Tcl syntax to represent Unicode, for example:<br>[encoding convert from utf-8 "A-Umlaut: \xc3\x84"] |
| SetParamInt | FullModuleName<br>ParameterName<br>Value | Command for setting an integer parameter. |
| SetParamFloat | FullModuleName<br>ParameterName<br>Value | Command for setting a floating point parameter. |
| SetParamEnum | FullModuleName<br>ParameterName<br>Value | Command for setting an addressed enum parameter. Value is the string of the enum value.<br>Example: Process0/module1 (SetSignalStatus), "SetParamEnum Process0/module1 Mode High" sets parameter Mode to High. |
| SetParamString | FullModuleName<br>ParameterName<br>Value | Command for setting a string parameter. |
| SetParamIntField | FullModuleName<br>ParameterName<br>Index<br>Value1<br>{OtherValues} | Command for setting integer field entries starting with index "Index" for the following values. |
| SetParamFloatField | FullModuleName<br>ParameterName<br>Index<br>Value<br>{OtherValues} | Command for setting float field entries starting with index "Index" for the following values. |
| SetParamStatic | FullModuleName<br>ParameterName | If a parameter can be static as well as dynamic, in the vast majority of cases the parameter is set to dynamic per default. Use this command to set such a parameter to static. |
| SetParamDynamic | FullModuleName<br>ParameterName | You can use this function to set a parameter to dynamic (if the parameter can be can be dynamic at all). |

| Command | Parameters | Description |
|---|---|---|
| SetLinkParam | FullModuleName<br>PortName<br>ParameterName<br>Value<br>{ OtherParameterName<br>OtherParameterValue } | Command for configuring links. The connection is identified via the connection node of a module. |
| SetResourceMapping | FullModuleName<br>ResourceName<br>[Index]<br>NewMapping | Command for changing the mapping of resources to a module. The resource ResourceName will be mapped to the new index NewMapping. In case the module allocates multiple resources of the same type, use the optional parameter Index for selecting the resource. |
| **Further Editing Commands** | | |
| Move | FullModuleName<br>[HierNodeName]<br>PosX<br>PosY | Command for re-positioning a module, a H-Box node, a comment box, or a simulation module to a new position within the design window.<br>Use parameter [HierNodeName] only for selecting an H-Box node. Nodes within hierarchical boxes are identified via the prefix "INBOUND#" or "OUTBOUND#"(like you do when using command ConnectModules), for example, "INBOUND#I000". |
| CreateSimSource | FullSimModuleName<br>xPos<br>yPos | Command for creating a simulation source on the indicated position. |
| CreateSimProbe | FullSimModuleName<br>xPos<br>yPos | Command for creating a simulation probe on the indicated position. |
| ConnectSimModule | FullSimModuleName<br>TargetModuleName<br>TargetModulePortName<br>[ xPos yPos ] | Command for connecting a simulation module with a link. The connection is identified via the specified module (TargetModuleName) and port (TargetModulePortName). TargetModuleName must be located in the same hierarchy as FullSimModuleName. Optionally, you can define the position of the connection on the link via the parameters xPos and yPos. |
| SetSimModuleProperty | FullSimModuleName<br>Key<br>Value<br>{OtherValue} | Command for configuring a simulation module. The following parameters can be set (in the list below, the key word Key is followed by the parameter values Value):<br>PixelAligmentOffset BITS<br>ImageFileSubPixels NUMBER<br>ImageFilePixelBits BITS<br>ImageFileNormalize CHOICE |

| Command | Parameters | Description |
|---|---|---|
| | | where CHOICE = "1" \| "0" \| "yes" \| "no" \| "on" \| "off" \| "true" \| "false"<br><br>CropRect [imageNr] X Y WIDTH HEIGHT<br><br>CurrentImage INDEX |
| AddImage | FullSimSourceName<br><br>ImageFileName1<br><br>{OtherImageFileName}<br><br>[ImageLibrary] | Command for adding an image to a simulation source. If the number of kernel elements on the link is > 1, you need to specify as many images as there are kernel elements on the link. Optionally, you can specify the image library you want to be used ( ImageLibrary = "Native" \| "ImageMagick"). |
| **User Libraries** | | |
| InstUserLibOperator | LibName<br><br>OperatorName<br><br>FullModuleName<br><br>XPos<br><br>YPos | Command for instantiating a user library element. |
| **Custom Library Handling** | | |
| InstCustomLibOperator | LibName<br><br>OperatorName<br><br>FullModuleName<br><br>XPos<br><br>YPos | Command for instantiating a custom library element. |

Table 4.10. Commands for Creating and Editing a VA Design in Tcl

## 4.13. Print / Screenshot

VisualApplets offers the possibility to print a diagram window or copy its content as an image into the system clipboard. The printing / screenshot is applied to the currently selected diagram window. The full window including all operators and links will be selected for output even if parts of the content cannot be fit into the current screen due to the limited resolution.

To start printing select the required diagram window and select **File** -> **Print** ( **Ctrl+P** ). The design will automatically be resized.

To generate a screenshot and copy the content into clipboard select the required diagram window and click on **Edit** -> **Screenshot to clipboard**. If you prefer to directly save the screenshot in an image file select **Edit** -> **Screenshot to file**.

## 4.14. Migration from Older Versions

In VisualApplets version 3, files from version 1.2 and higher can be loaded.

Loading of files from previous versions might result in incomplete designs. Some names, functionality and parameters of some links and operators might have changed. VisualApplets will adapt, add, and transform most of the module settings to the new parameter sets. However, in some cases, an automatic transformation cannot be performed. In any case, you should verify your implementation if migrated from a previous version.

Migration from older versions can result in one of the following:

- **"Update to Current Version" Window**

  When a design using operators from an older VisualApplets version is opened, VisualApplets might ask for an update of the design to the current version.

- **Open Input Links at Signal Operators**

  If you do not use the auto update function "Update to Current Version", the inputs of some signal operators might not be connected. VisualApplets offers a function to automatically connect *Reset* inputs to operator *Gnd* and *Tick* inputs to operator *Vcc*. Simply right click on the respective operator and select **Connect to ground** or **Connect to Vcc**.

- **Missing Operators**

  If an operator which was previously used does not exist in the current version anymore, it is replaced by operator *Dummy*. The missing operator has to be replaced. For most missing operators a replacement is available. Read the documentation of the operator in detail in Section 20.11, 'Dummy'.



**Release Notes**

Check the VisualApplets release notes to get more information about changes.

# 5. Embedded VisualApplets (eVA)

## 5.1. Introduction

**Embedded VisualApplets** allows you to use the graphical FPGA development environment VisualApplets for programming machine vision applications that will run on the FPGAs on your hardware platforms.

You do not program the whole FPGA with VisualApplets, but only an IP Core (Intellectual Property Core). This IP Core is **embedded** in your surrounding FPGA design:

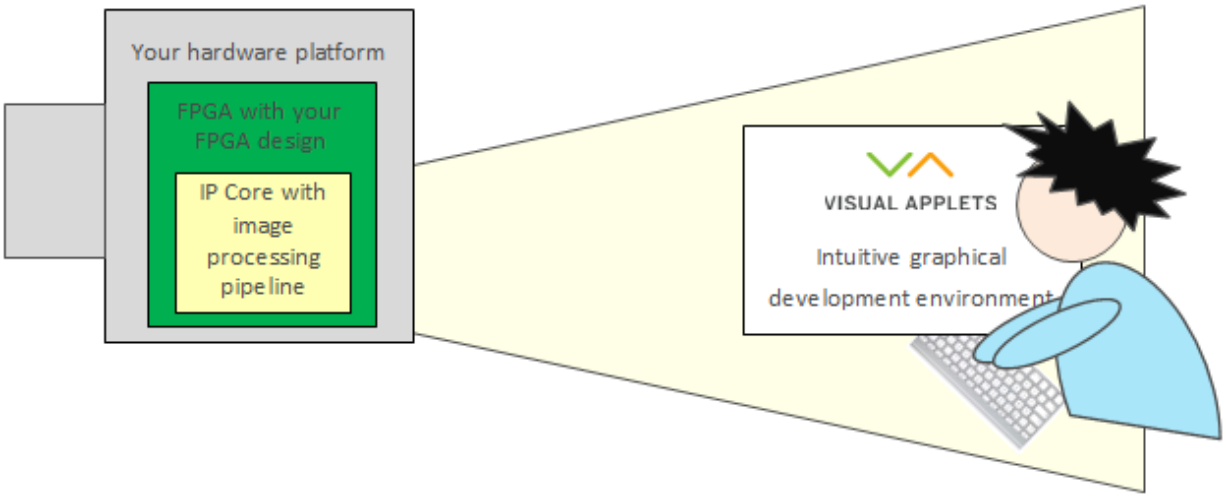



Figure 5.1. Graphical Programming of Image Processing Applications on FPGAs

To fill the IP core with logic for an image processing pipeline, no HDL knowledge is required, as VisualApplets is a graphical environment. VisualApplets also cares for the entire implementation flow. With VisualApplets, you enable software engineers and image processing experts to program the IP core on your hardware.

Before you can use VisualApplets to program applications for the FPGA on your hardware, you need to integrate the IP Core into your FPGA design and to generate an eVA (embedded VisualApplets) Plugin that provides VisualApplets with all hardware-specific details of your hardware platform.

### 5.1.1. Integration Workflow

You need to integrate the VisualApplets (VA) IP Core once. VA IP core: IP core in your FPGA that you can program with image processing functionality via VisualApplets. After integration, you can re-program the VA IP core as often as you want with as many machine vision applications as you want.

The key steps for hardware and software integration are carried out automatically by the tool **eVA Designer** (which comes as part of the VisualApplets Embedder package). This speeds up the work flow and leads to an implementation which is correct by construction.

You integrate VisualApplets into your hardware design in just a few steps:

- You install VisualApplets.
- You let **eVA Designer** generate an IP core black box in VHDL. The intuitive GUI supports you in specifying the details of the future VA IP core:
  - You enter some data regarding your hardware (hardware name, FPGA type, your vendor name etc.).

- You specify the ports you want the VA IP core to have (image input ports, image output ports, register interfaces, memory interfaces, GPIs and GPOs).
- You let **eVA Designer** generate the empty VA IP Core (VHDL black box) automatically, based on your inputs.

- You integrate the generated IP core black box into your FPGA design (VHDL).
- You generate a netlist of your FPGA design with integrated black box.
- You create a constraints file.
- You provide the netlist and the constraints file to **eVA Designer** and let it generate the hardware-platform-specific **eVA Plugin Installer** for VisualApplets.
- You execute the **eVA Plugin Installer**. After installation, the hardware-platform-specific **eVA Plugin** is available in VisualApplets and allows to develop designs for the VA IP core on your FPGA.

After proceeding these steps, you can use the graphical development environment VisualApplets for programming the VA IP Core on your FPGA. A detailed step-by-step guide for these steps you find in Section 5.3, 'Defining the IP Core Properties'.

Once-only implementation process for a new hardware platform:

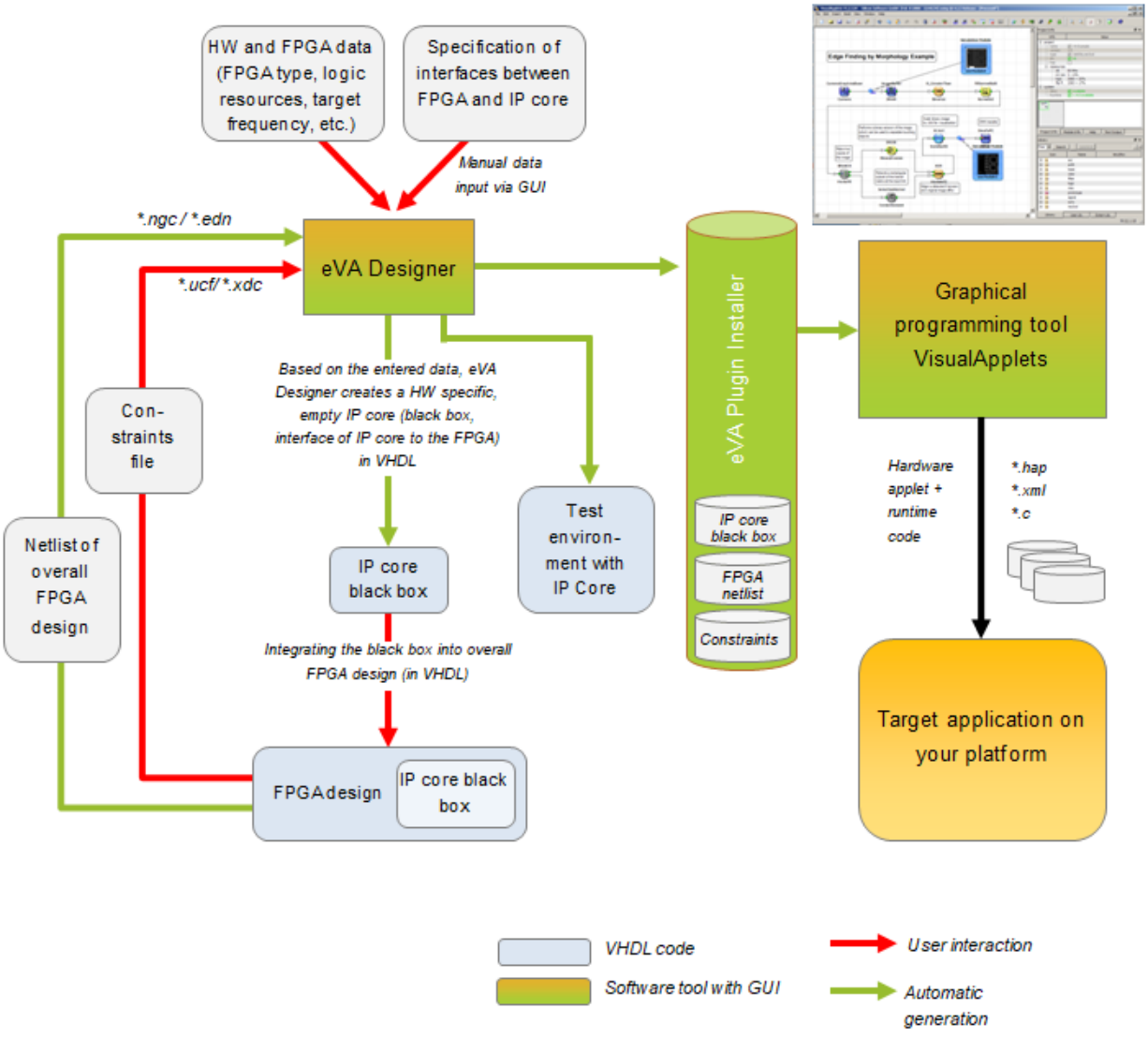



Figure 5.2. Once-Only Integration Process for new Hardware Platform

**Your FPGA design as a building block:**

The netlist of your camera's FPGA is – via the interfaces of the black box – connected to the variable image processing designs created with VisualApplets:

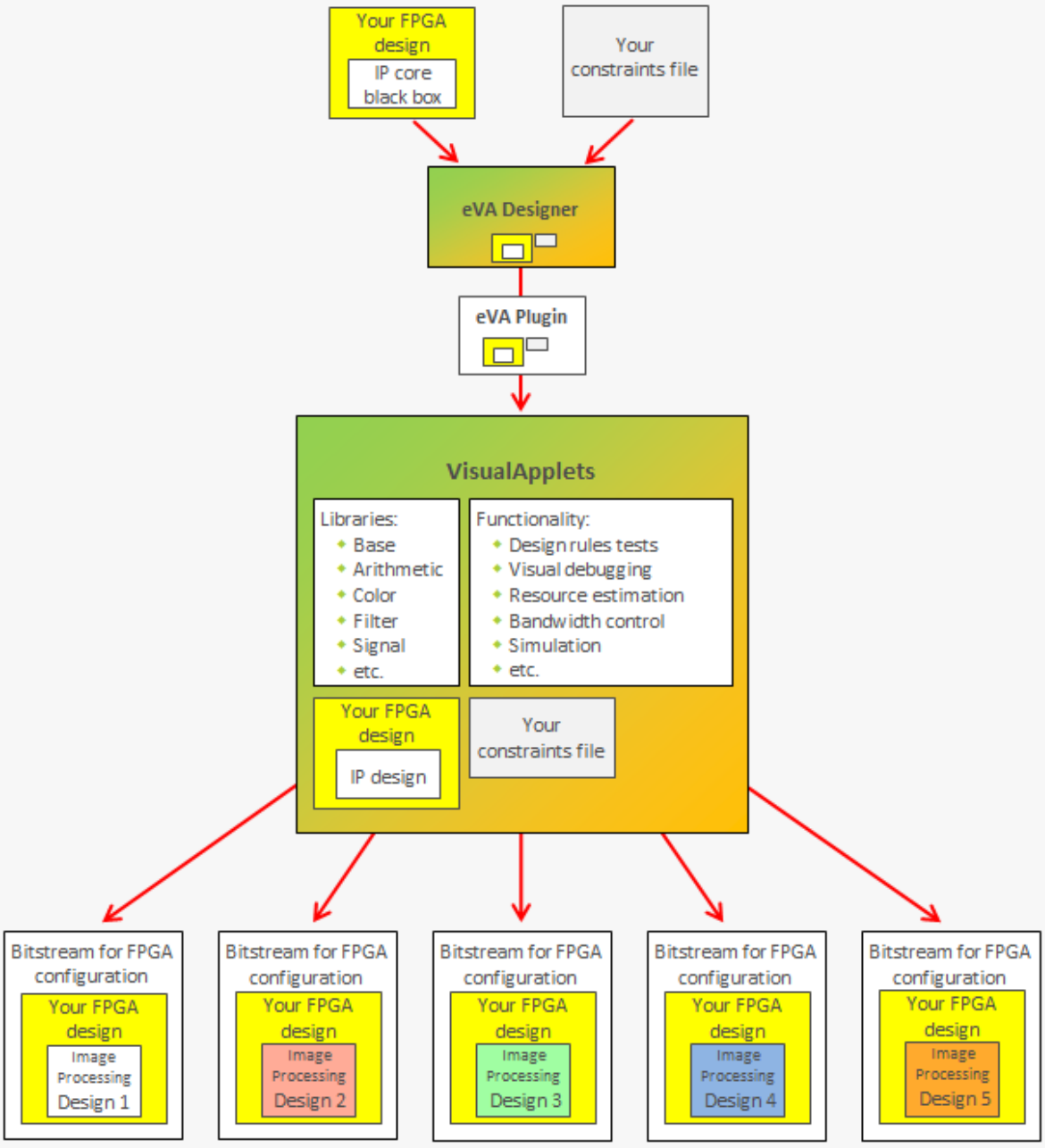



Figure 5.3. FPGA Design and IP Core Content as Building Blocks for Bitstream Generation

### 5.1.2. Have a Glance at VisualApplets

By implementing the VA IP Core into your FPGA design, you enable software engineers and image processing experts to program the VA IP Core on your hardware.

VisualApplets is used to design image processing programs (applets) for FPGA-based image processing. Designing applets with VisualApplets is easy. It is done in a graphical development environment. No knowledge of any hardware description language is necessary. In VisualApplets, an image processing solution is developed in form of a flow chart – without any HDL.

This is how a design looks like in VisualApplets:

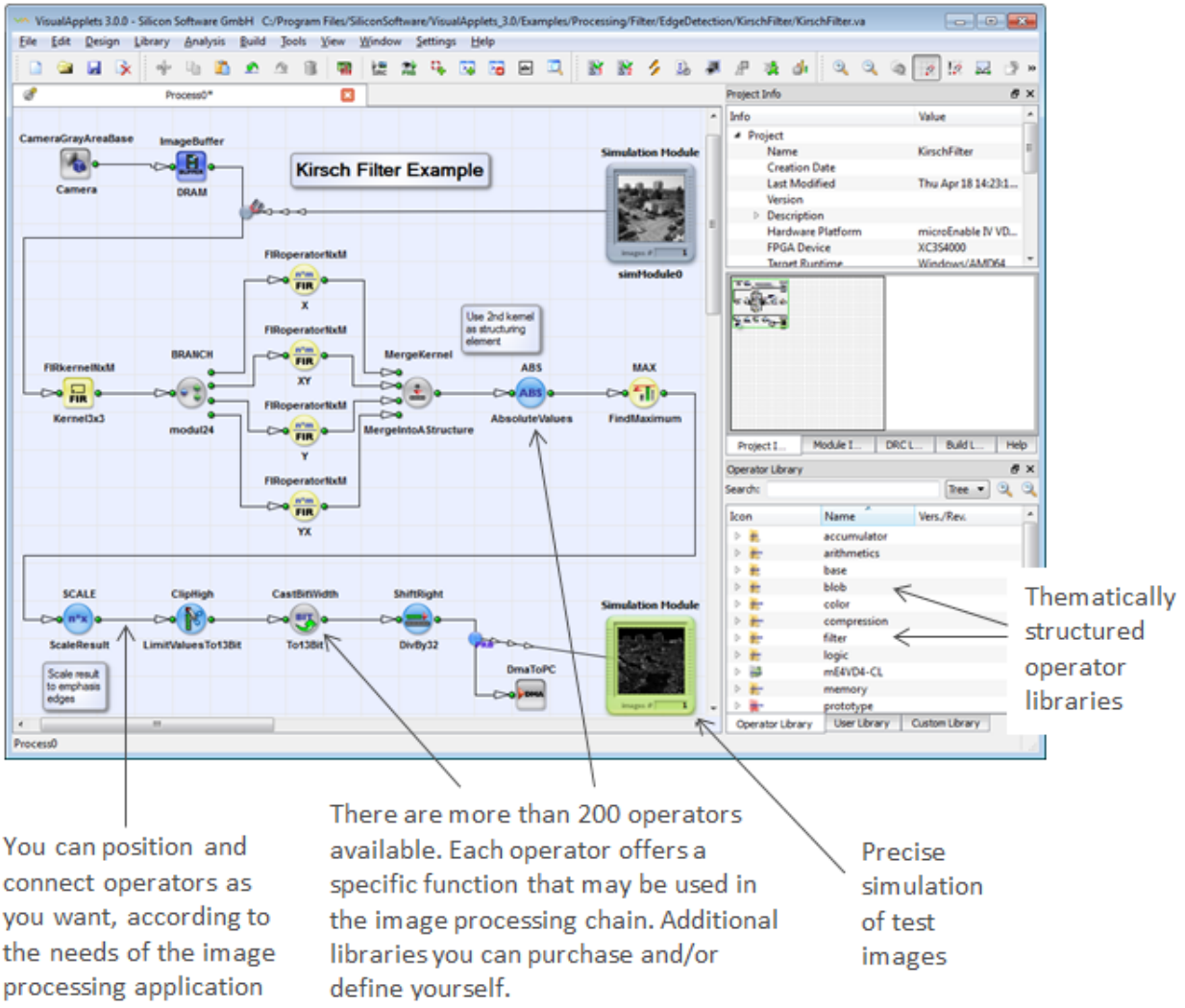



Figure 5.4. VisualApplets Program Window with Image Processing Design

Designs like in the figure above will later on (after synthesis into a bitstream) populate the VA IP Core on your hardware.

VisualApplets provides more than 200 operators. Each operator offers a specific function that may be used in the image processing chain.

Some of the operators need to connect directly to the surrounding FPGA environment. These are:

1. Operators that receive images from the camera/sensor
2. Operators that deliver processed images at the end of the processing pipeline (e.g., operators delivering processed images to DMA)

3. Buffering operators using RAM resources
4. Operators allowing signal input and output (GPIOs)

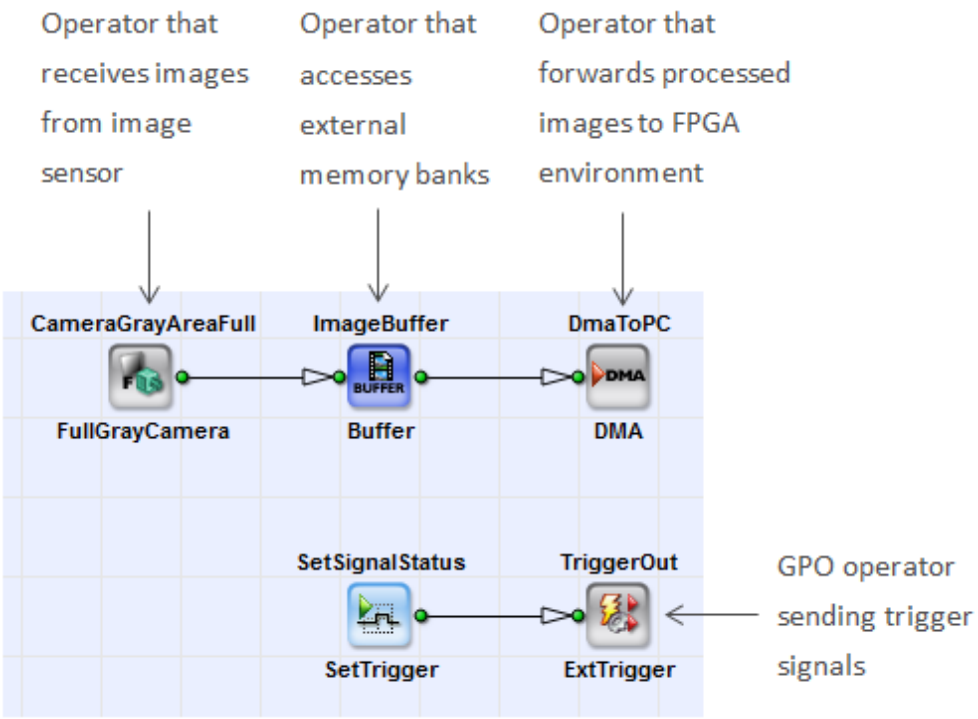



Figure 5.5. Example for a Simple Image Acquisition Applet with Interface-Requiring Operators

For data communication between these operators and the surrounding FPGA design, the VA IP Core needs to provide according interfaces.

### 5.1.3. Concept of IP Core Interfaces

You can configure the interface ports of the VA IP core with a high grade of flexibility. The I/O of a VA IP core is composed of a number of flexible and easy-to-use interfaces between the IP core and your (surrounding) FPGA design. External hardware resources (like sensor interface, memory controller, etc.) are linked by glue logic in your FPGA design as shown in the figure below:

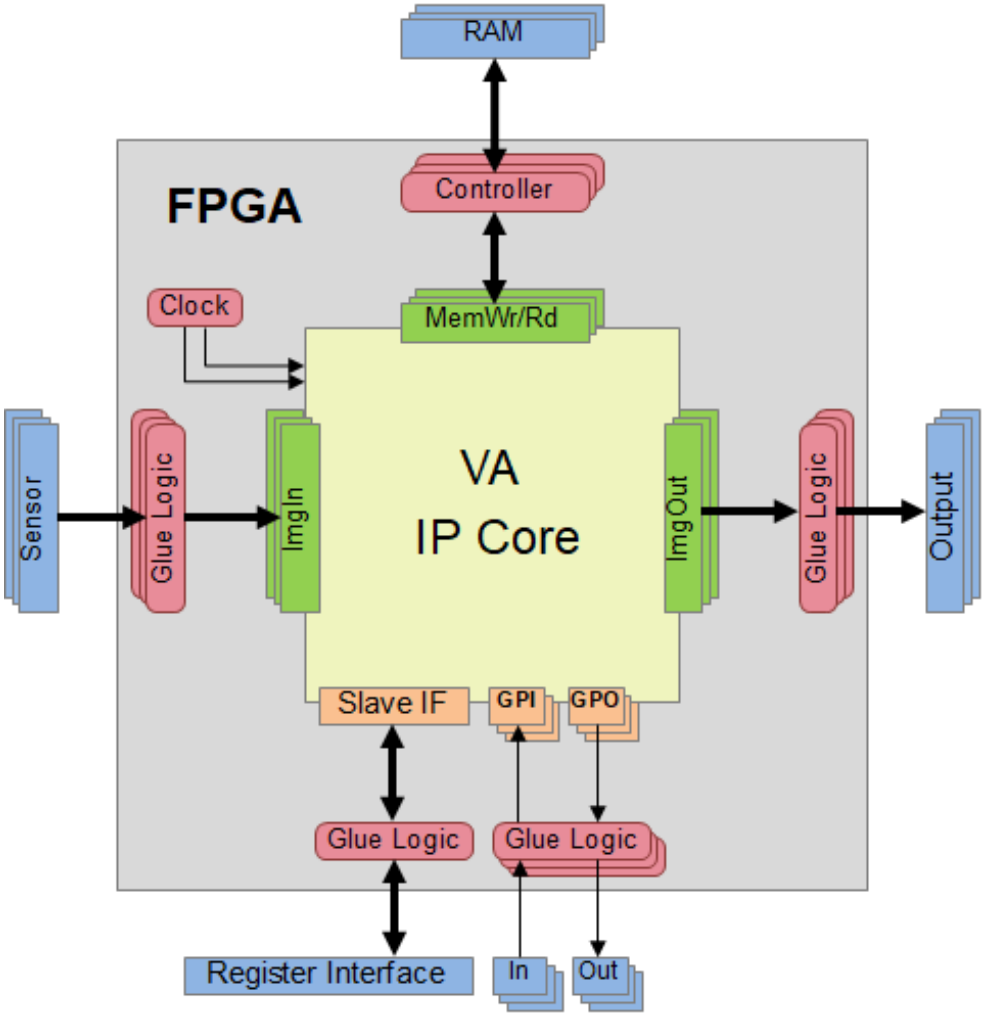
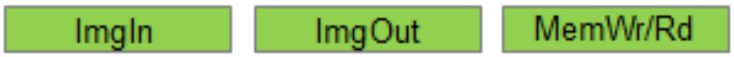



Figure 5.6. Concept of VA IP Core Interfaces

Interfaces that use a first-in-first-out buffer (FIFO) are marked green:




There are the following types of interfaces:

- **Clock:** The IP core runs with two phase-synchronous design clocks where the second clock has twice the frequency of the first clock.
- **ImgIn: Interfaces for input streaming of image data** Image data enters the IP core via a simple FIFO interface where additional flags for end-of-line and end-of-frame mark the frame boundaries. The number of ImgIn ports as well as the layout of data communicated via these ports can be configured.
- **ImgOut: Interfaces for output streaming of image data** Image data leaves the IP core via a simple FIFO interface where additional flags for end-of-line and end-of-frame mark the frame boundaries. The number of ImgOut ports as well as the layout of data communicated via these ports can be configured.
- **MemWr/Rd: Interfaces for connecting external memory** The IP core may be connected to external memory via an abstracted memory interface where any kind of memory can be connected via a single interface mechanism. External glue logic needs to adapt the IP core's memory interface protocol to the used memory controller. The number of available memory interface ports can be configured. For each port, the interface properties may be defined individually (i.e., address and data width).

- **GPI, GPO : General purpose signal I/O** Via GPI/GPO, signals may enter or leave the IP core. Such signals can be used for triggering and process control. The number of GPIO signals can be configured.
- **Slave IF: Register slave interface for runtime access to design parameters** The slave interface is a simple register interface controlled by address, data, and control signals synchronous to the design clock. Controlling Reset and Enable of the implemented image processing pipeline(s) in the IP core is also done via the slave interface.

### 5.1.4. Performance Classes

VisualApplets Embedder supports 12 performance classes. The individual performance classes are defined by the maximal sensor bandwidth, the FPGA resources that can be maximally used by the image processing application, and by the option to define memory interfaces:

| | Bandwidth Classes | 150 MB/s | 500 MB/s | 1000 MB/s | unlimited |
|---|---|---|---|---|---|
| Economy *Moderate Resources, no DRAM Interfaces* | Performance Class | E150 | E500 | E1000 | EL |
| | Sensor Bandwidth | Up to 150 MB/s | Up to 500 MB/s | Up to 1000 MB/s | unlimited |
| | Available Resources | Up to 50.000 LUT4 | Up to 100.000 LUT4 | Up to 150.000 LUT4 | Up to 200.000 LUT4 |
| | DRAM Interfaces | Not supported | Not supported | Not supported | Not supported |
| eXtended *High Resources and Support of DRAM* | Performance Class | X150 | X500 | X1000 | XL |
| | Sensor Bandwidth | Up to 150 MB/s | Up to 500 MB/s | Up to 1000 MB/s | unlimited |
| | Available Resources | Up to 100.000 LUT4 | Up to 300.000 LUT4 | Up to 500.000 LUT4 | Up to 750.000 LUT4 |
| | DRAM Interfaces | Supported | Supported | Supported | Supported |
| Superior *Maximum Resources and Support of DRAM* | Performance Class | S150 | S500 | S1000 | SL |
| | Sensor Bandwidth | Up to 150 MB/s | Up to 500 MB/s | Up to 1000 MB/s | unlimited |
| | Available Resources | unlimited | unlimited | unlimited | unlimited |
| | DRAM Interfaces | Supported | Supported | Supported | Supported |

You select the performance class. For pricing issues and licensing procedure, see section Section 5.6, 'Licensing Model'. You need for a specific target hardware.

For entering data to the XML file (=hardware description file), you should use the GUI tool eVA Designer as described in section Defining the IP Core Properties. For each performance class, one XML template is available. You use the XML template to enter

- the hardware specifics of your hardware platform,
- a description of the interfaces you need at the VA IP Core,
- the hardware-specific VisualApplets operators that connect to the interfaces from within the VA IP core.



**Available Templates**

Your VisualApplets installation comes with two example platform descriptions (XML), as well as with one template that allows you to enter the details of your own hardware from scratch. These XML files are for **testing purposes**. They allow the generation of restricted eVA Plugins. Applets created with these plugins are runtime-limited. The XML example platform descriptions and template you find in your runtime installation, folders:

- `VisualApplets_<version_number>\Examples\EmbeddedVisualApplets\DemoTemplate`
- `VisualApplets_<version_number>\Examples\EmbeddedVisualApplets\SVDK`
- `VisualApplets_<version_number>\Examples\EmbeddedVisualApplets\ZC702`

With your own, vendor-specific XML templates for each performance class you will be provided directly by Basler.

### 5.1.5. Requirements

To integrate the VA IP Core into your FPGA design and to generate an eVA plugin that provides VisualApplets with all hardware-specific details of your hardware platform, you need the following components:

Hardware & Operating System:

- PC running operating systems Microsoft Windows 7, Windows 8, or Windows 10 (64bit)
- PC Memory: Minimum 4 GByte, recommended: 8 GByte or better
- Minimum available hard disk space: 500 MByte
- Target hardware platform with implemented FPGA. At the point of release of VisualApplets 3.2, FPGAs of Xilinx Inc. starting from series 6 are supported (including Zynq7000, Series 7 FPGAs, Ultrascale, Ultrascale+, and Zynq Ultrascale+).

Software:

- VisualApplets 3.0 (or higher)
- VisualApplets 3 license
- VisualApplets 3 Embedder license
- XML template for creating a hardware description (delivered by Basler)

Third-Party-Software for synthesizing image processing applications:

- For compiling the SDK examples, a C++ Compiler is necessary.
- For generating the actual bitstream out of your design, you need a tool suite provided by the FPGA manufacturer. The tool flow is completely controlled by VisualApplets. You simply need to install it. Depending on the FPGA type you have implemented, you need (at point of release of VisualApplets 3.0) either

  1. Xilinx Vivado (feeless WebPACK edition or Design Suite), or
  2. Xilinx ISE (feeless WebPACK edition or Design Suite).

  For details, please refer to the documentation of the FPGA type you have implemented on your hardware.

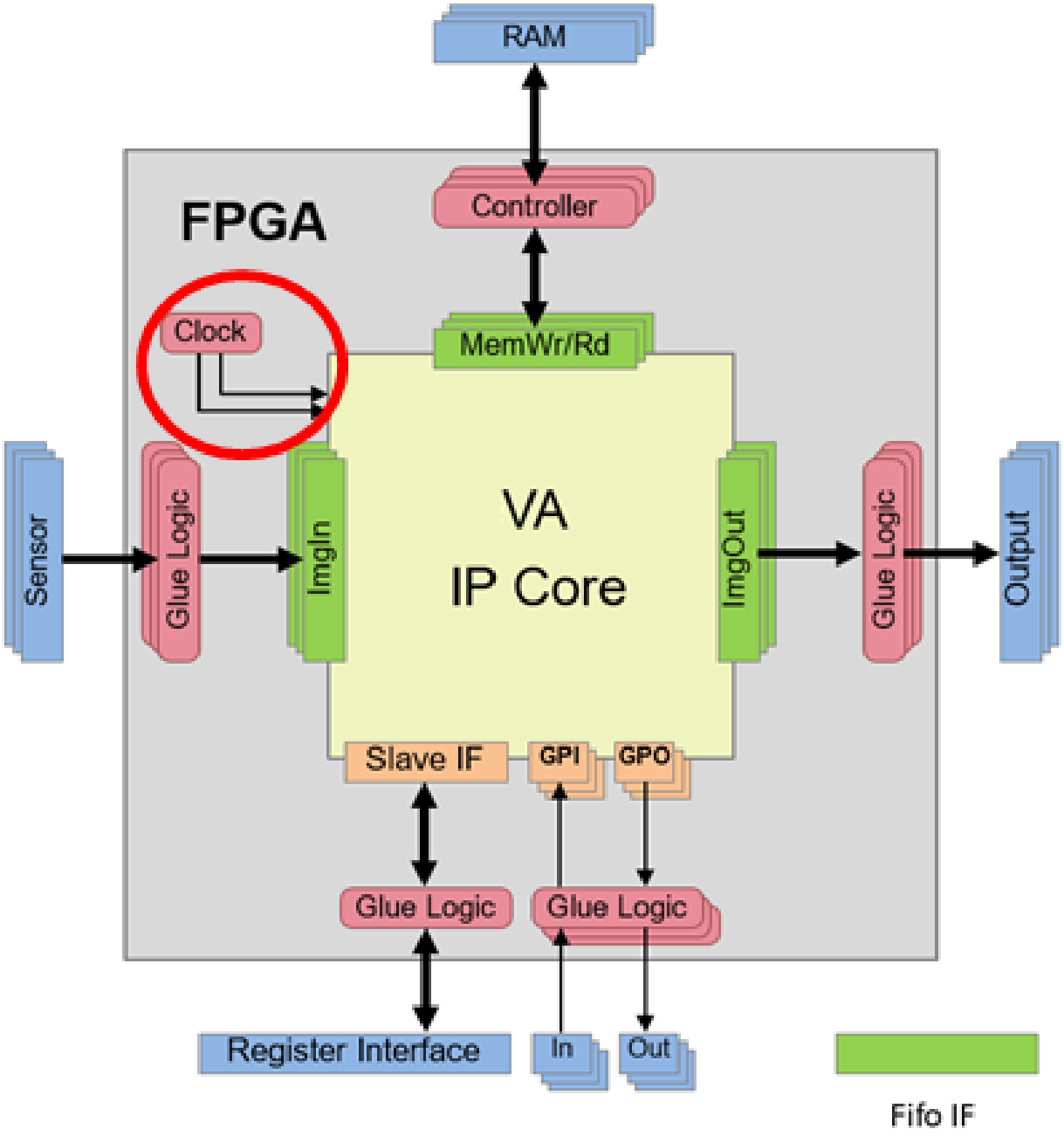
## 5.2. Common Interfaces for all Platforms

You define number and type of the interfaces of the VA IP core (as described Section 5.3, 'Defining the IP Core Properties'.

However, two Interfaces are the same for all platforms: The clock interface and the register slave interface. These will be described here before you actually start to fill in your hardware description file.

### 5.2.1. Clock Interface

In hardware, you need to provide two clock inputs to the IP core: *iDesignClk* and *iDesignClk2x*. Such clock signals you can usually generate quite easily using a digital clock manager of the FPGA (under your control).




These clocks must hold following requirements:

- Being phase synchronous
- Having a frequency ratio of 2 where *iDesignClk2x* is the faster one.

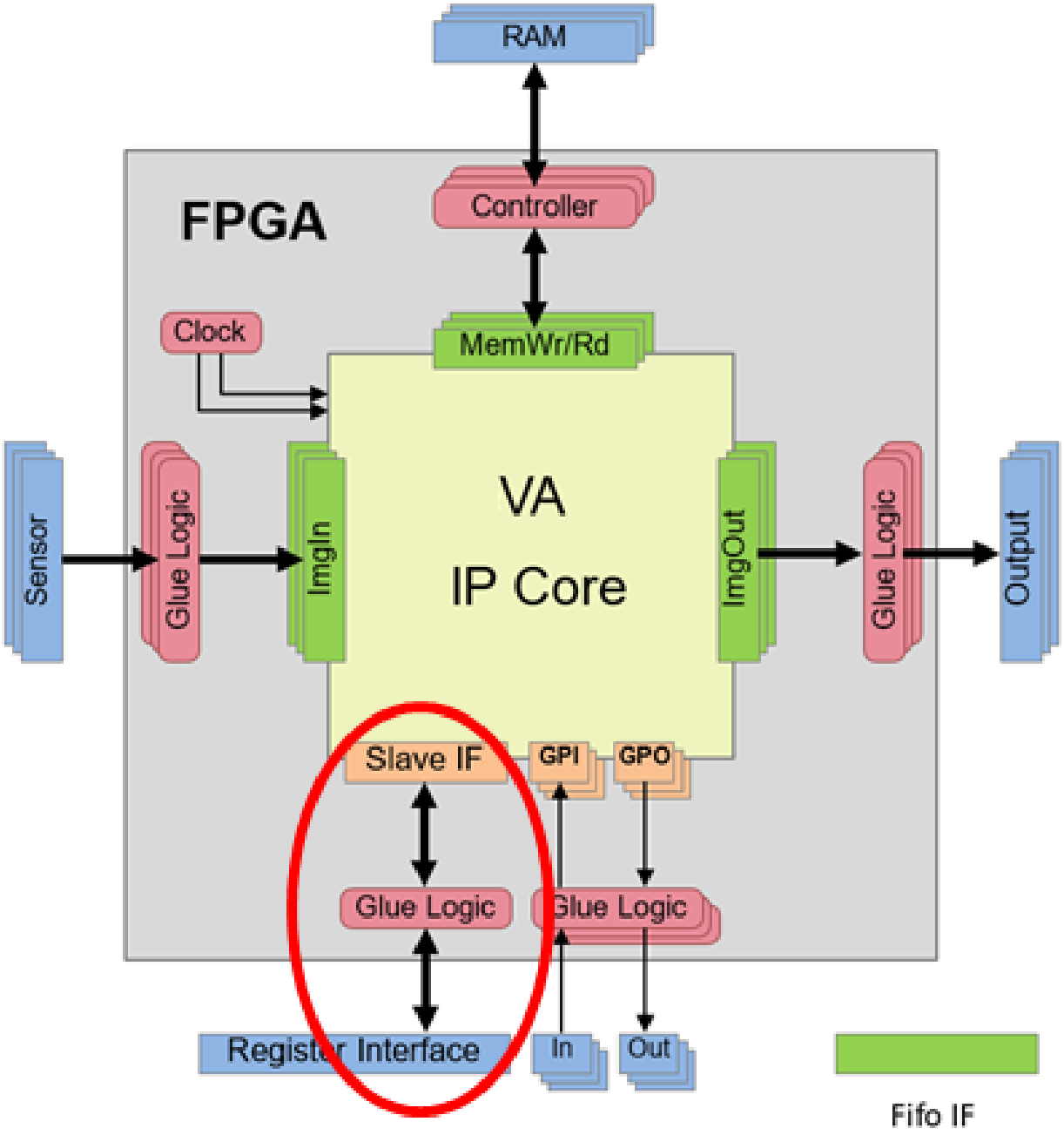
**Influence of clock input on design speed:** All operators of a VisualApplets design work synchronous to the rising edge of *iDesignClk*.

During IP core definition in **eVA Designer**, you will set up an allowed frequency range constraint for the IP core (i.e., for *iDesignClk*) (as described in section Section 5.3.4, 'Entering FPGA Details', step 3). This information is provided to VisualApplets. During design with VisualApplets, the VisualApplets

user selects a target frequency (for *iDesignClk*) within the allowed range. This information will be used for synthesis and implementation: After synthesis of the VisualApplets design, the target frequency is stored in the `*.hap` file (that also contains the synthesized FPGA bitstream). The runtime system contains a function for acquiring the target design frequency from the `*.hap` file.

### 5.2.2. Register Slave Interface

The Register Slave Interface of the VA IP Core is a simple register interface controlled by address, data, and control signals synchronous to the design clock.




Dynamic design parameters (i.e., dynamic parameters of operator instances in an image application) are communicated via the register slave interface. The interface provides read and write operations.

The register interface has the following properties:

- I/O is synchronous to clock *iDesignClk*.

- Reset and Enable signals have no effect on register values.
- The data width is 32 bit.
- The address width of the interface is 16 bit.
- Write and read may occur simultaneously.
- Registers which are accessed through the register interface may have any width between 1 and 64. Mapping between the slave interface data width and the actual register width of a VisualApplets parameter is done automatically. When the width of a parameter register is bigger than the width of the register interface the runtime software will divide the access automatically. A single parameter then consumes more than one register address.

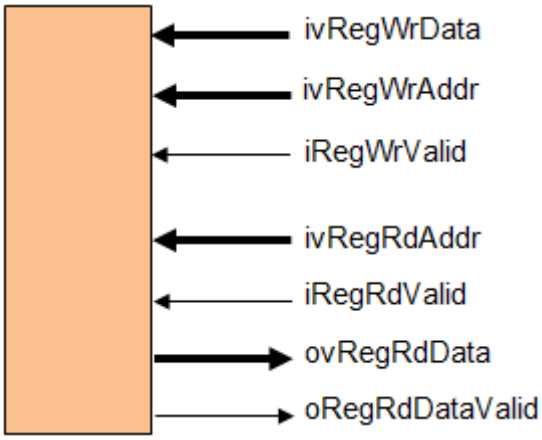



Figure 5.7. Ports of the Register Interface

The figure above shows the ports related to the register interface.

**Writing:** The communication protocol for writing is based on the write address *ivRegWrAddr*, the write data *ivRegWrData* and the write strobe *iRegWrValid*. Any assertion of the high active signal *iRegWrValid* initiates a single write access to a VisualApplets register with the address *ivRegWrAddr*. Write accesses can be done in subsequent clock ticks so implicit bursts are allowed.

**Reading:** The communication protocol for reading is based on the read address *ivRegRdAddr*, the read strobe *iRegRdValid*, and the valid signal for read data *oRegRdDataValid* where the data is then output at the port *ovRegRdData*. Any assertion of the high active signal *iRegRdValid* initiates a single read access from a VisualApplets register with the address *ivRegRdAddr*. The latency of the returned output value may depend on the configuration of the eVA IP Core and the VisualApplets design.

When running a VisualApplets design in the VA IP Core on your hardware platform, the parameters are accessible via the VisualApplets runtime software interface.

VisualApplets supports different models for accessing design parameters at runtime:

- **Using an eVA runtime environment based on HAP files** This approach is very similar to the runtime interface of frame grabbers from Basler. The HAP file contains all necessary information for accessing design specific parameters easily. A runtime software API is provided which can load HAP files, extract the FPGA configuration data, and provide access to the design parameters.
- **Using GenICam API based on generated GenICam XML code** When the target platform is connected via a GenICam compatible interface to the software this option allows a seamless integration of image processing parameters into the GenICam API. There is no need for any additional software component.
- **Using generic, design- specific C API code generated by VisualApplets** The generated code is platform independent ANSI-C code. Callback functions for write and read access to registers via

the VA IP Core register slave are registered at startup. Then the code provides access to parameters addressed by name where translation functions trigger the communication via the call back functions. This approach is well suited for software integration in embedded systems (e.g., Zynq7000).



**Automatic Generation of Runtime Interface Files**

The files for a design-specific runtime interface are automatically generated by VisualApplets (for all three access models).

The options for the runtime software interface are described in detail in Section 5.5, 'Runtime Software Interface'.

### 5.2.3. Reset and Enable

The VA IP Core uses the register slave interface also for controlling the *Reset* and *Enable* signals – instead of using dedicated ports for that purpose. This way, *Reset* and *Enable* are under user control via the runtime software, and a not a priori known number of processes can be controlled separately.

The *Reset* and *Enable* signals interact with the VA IP Core design as follows:

- Each process has its own *Reset* and *Enable* signals (high active) which are set by 1-bit registers.
- Assertion of *Reset* puts the complete process in its init state.
- Assertion of *Enable* starts processing.
- Deactivating *Enable* stops processing. When *Enable=0*, input FIFOs are not read. Regarding output FIFOs, depending on the state of the image processing pipeline some data still may be written to them after deasserting *Enable* but the flow control safely prevents that any FIFO content gets corrupted.
- *Reset* may only be asserted when *Enable=0*.
- *Reset* will empty all FIFOs.
- *Enable* has effect on any hardware specific operator with FIFO interface.
- *Reset* and *Enable* have no effect on the parameters of the operators.
- *Reset* and *Enable* have no effect on the GPIO ports.
- For any *ImgIn* and *ImgOut* interface of the IP core, there are the output ports *ResetO* and *EnableO*. Those outputs are connected to the according *Reset* and *Enable* signals of the process where the interface is used. Any logic attached to the concerning interface is therefore able to perform an appropriate action during the reset and disable state.
- After deasserting *Enable*, the design needs to be reset before it is enabled again. Setting *Enable = 0* and subsequently *Enable = 1* without asserting *Reset* may lead to unpredictable behavior for some VisualApplets designs.

Recommended START procedure for a process:

1. Send *Reset*.
2. Assert *Enable*.

Recommended STOP procedure of a process:

1. Deactivate *Enable*.
2. Send *Reset*.

Any read or write access to the FIFO of any platform specific operator is enabled by the *Enable* signal of the concerning process (with some design dependent latency caused by the flow control). Also the *Reset* signal of a process is connected to the reset port of any I/O FIFO associated with that process.

## 5.3. Defining the IP Core Properties

For defining the interfaces of the individual VA IP Core you want to use on your FPGA, you use the GUI tool **eVA Designer**. eVA Designer is part of the VisualApplets Embedder package. eVA Designer is installed together with VisualApplets. You can use eVA Designer after acquiring a VisualApplets Embedder license. The data you enter here will be written to the hardware description file (XML). Out of this data, **eVA Designer** will later on build the IP Core black box in VHDL.

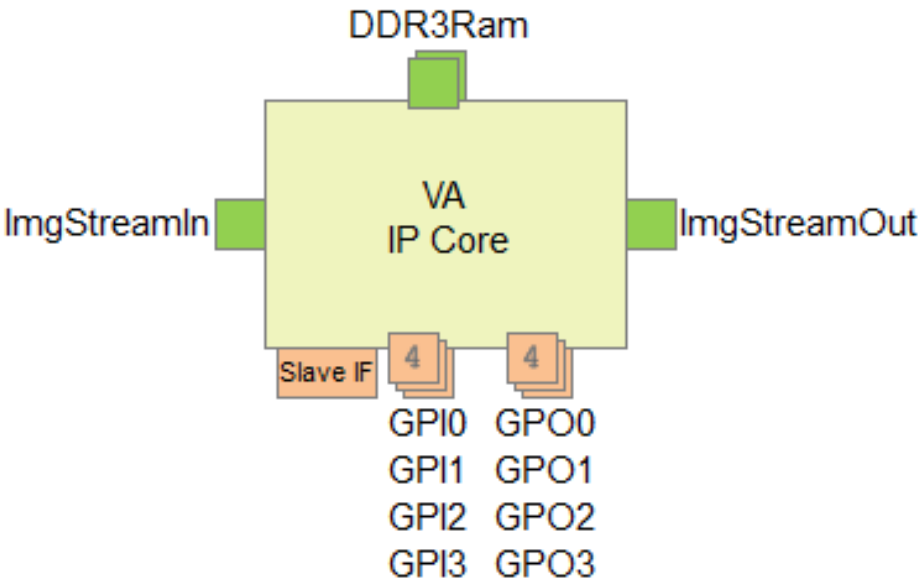
### 5.3.1. Graphical User Interface of eVA Designer

The graphical user interface of **eVA Designer** is easy and intuitive:

The GUI of **eVA Designer** supports you as it reacts to any of your inputs.

#### 5.3.1.1. Graphical Representation

The graphical representation of the defined IP Core (in the left hand panel of the program window) displays all interfaces you have defined at a given moment. The graphical representation reacts to any input (adding or deleting interfaces) immediately.




- All interfaces using FIFOs are displayed in green.
- Multiple interfaces of the same kind are displayed accordingly:
  - Up to three interfaces of the same kind are displayed as a pile of the exact number of interfaces. Example: 2 memory interfaces:

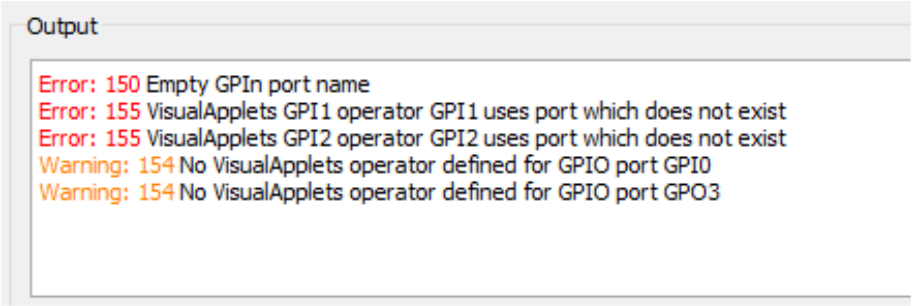



  - More than three interfaces of the same kind are displayed as a pile of three with a figure informing about the actual number of interfaces. Example: 4 GPI and 4 GPO interfaces:




### 5.3.1.2. Output field

The output field immediately informs you if your IP core definition is not congruent with other entries you make into **eVA Designer**. e.g., if the interfaces you defined don't match the VisualApplets operators you defined for connecting to these interfaces within the IP core.

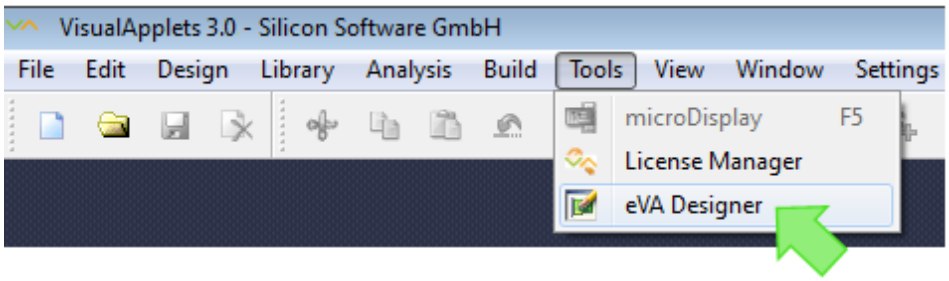



The output field reacts to any input (like adding or deleting interfaces or operators) immediately.

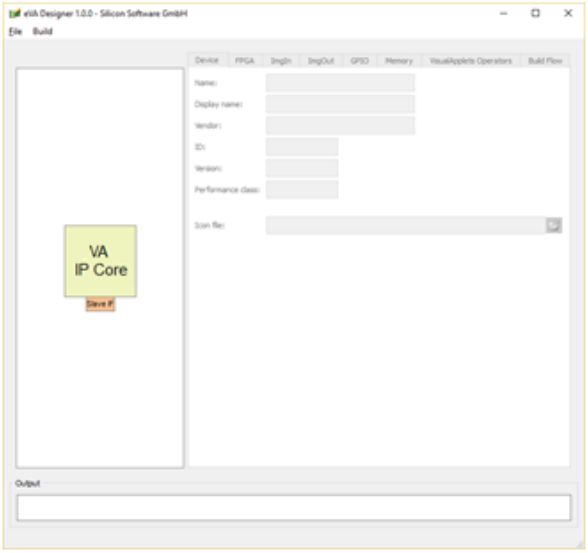
## 5.3.2. Opening eVA Designer and Hardware Description File

To create a new IP core definition, i.e., to define the interfaces of the programmable IP core you want to integrate into your FPGA design:

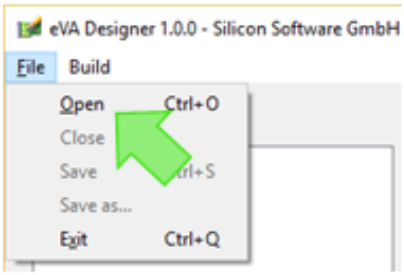
1. Start **VisualApplets**.
2. From the **Tools** menu, select **eVA Designer**.




The start window of **eVA Designer** opens:

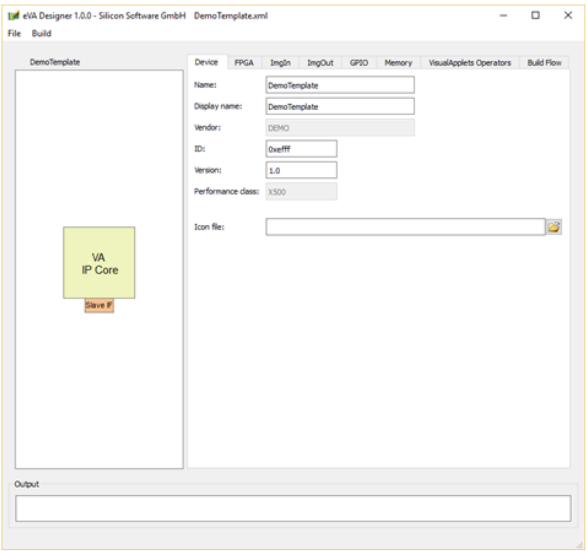



3. From the **File** menu, select **Open**.




4. Select an XML template (or an HW description file you have been working on earlier) from your file system to be loaded into **eVA Designer**.

   The XML template is now loaded into **eVA Designer**:





**Continuing Work on a Hardware Description File**

You can of course also re-open individual hardware description files you have already been working on, or one of the fully filled-in example files. In this case, the graphical representation of the IP core shows the interfaces that have already been specified:

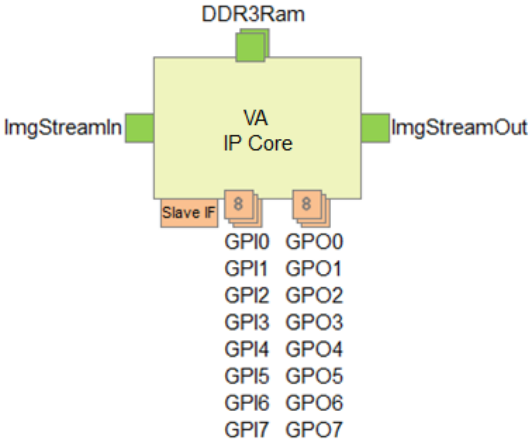



Figure 5.8. Example IP Core as specified for Zynq Platform

## 5.3.3. Entering Platform Details

1. Go to the **Device** tab.



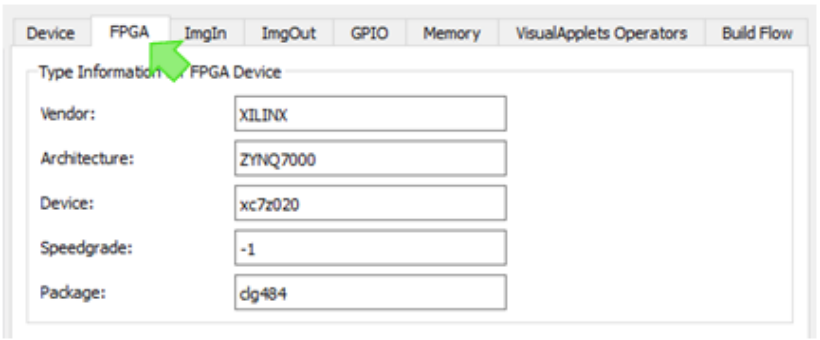

2. Enter first information regarding the hardware that contains the FPGA you want to equip with image processing applications:

   - **Name:**Enter the name of your target hardware product.
   - **Display Name:** Define an alias name for your hardware that you want to display in the VisualApplets GUI. Space characters are allowed.
   - **Vendor (display field):** Name of your company (i.e., the product's vendor). The value of that string is preset by Basler and cannot be changed. The vendor name will be visible in the VisualApplets graphical user interface. (The example HW description files show *DEMO* in this place.)
   - **ID:** Enter a four-digit hexadecimal identification number of your hardware platform which can be read out via the runtime system. You are free to enter any value here. The value is intended for hardware identification by the VisualApplets user. Example: *0xabcd*
   - **Version:** Enter the version of your hardware product. You are free to enter any value here. The value is intended for HW version identification by the VisualApplets user.

- **Performance Class:** Here, the performance class of the XML template you selected is displayed. For details regarding the available performance classes, see section Performance Classes.
- **Icon File (optionally):** You can enter here the path to a small graphics file that will display later in VisualApplets together with the name of your hardware platform.

### 5.3.4. Entering FPGA Details

1. Go to the **FPGA** tab.




2. Enter information regarding the FPGA you want to integrate an IP core in under **Type Information for FPGA Device**:

   **Vendor:** FPGA Vendor name (e.g., *XILINX*).

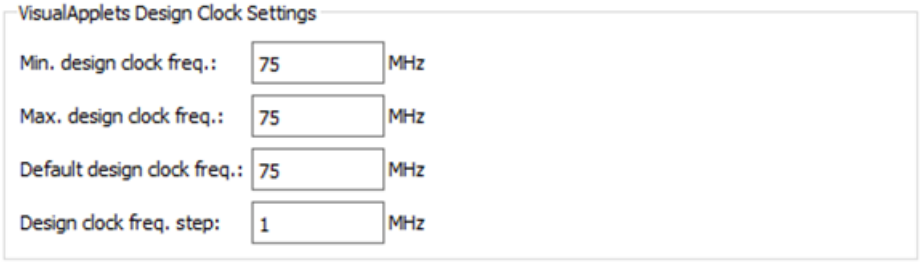
   **Architecture:** FPGA series and FPGA model of your FPGA, e.g., *ZYNQ7000*.

   **Device:** Device name (FPGA type) of your FPGA (without speed grade and package details), e.g., *xc7z020*.

   **Speedgrade:** Speed grade of your FPGA, e.g., *-1*

   **Package:** Package identification of your FPGA, e.g., *clg484*

3. Under **VisualApplets Design Clock Settings**, specify the properties of *iDesignClk*:




   **Min. design clock freq.:** Minimum frequency of the Visual Applets Core design.

   **Max. design clock freq.:** Maximum frequency of the Visual Applets Core design.

   **Default design clock freq.:** Default frequency of the Visual Applets Core design.

   **Design clock freq. step:** Step size by which the frequency of the Visual Applets Core may be adapted.

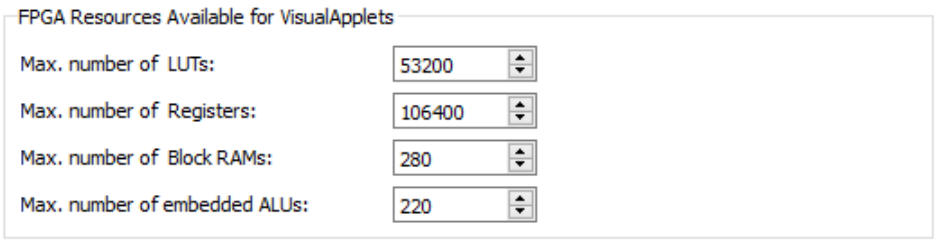
   If you specify a frequency **range** here (instead of entering the same value in all three upper fields), you allow the VisualApplets user to set the design clock for a specific design to a value within your value range. See Section 5.2.1, 'Clock Interface' for general information on the clock system.

**Surrounding FPGA Design Needs to Support the Defined Frequency**

Make sure your surrounding FPGA design provides the frequency you define here for the IP Core design. If you allow the VisualApplets user to adapt the design clock (by specifying a frequency **range** here instead of entering the same value in all three fields), you need to make sure that

- your surrounding FPGA design has the flexibility to adapt its input to *iDesignClk*, and
- your FPGA design is able to provide all frequencies within the specified value range.

4. Under **FPGA Resources Available for VisualApplets**, enter which FPGA resources you maximally allow an image processing application (in the VA IP Core) to use:




**Max. number of LUTs:** Maximum number of FPGA LUTs which may be used by the Visual Applets design.

**Max. number of Registers:** Maximum number of FPGA registers which may be used by the Visual Applets design.

**Max. number of Block RAMs:** Maximum number of block rams of the FPGA which may be used by the Visual Applets design.

**Max. number of embedded ALUs:** Maximum number of embedded ALUs of the FPGA which may be used by the Visual Applets design.

## 5.3.5. Entering Descriptions of Required *ImgIn* Interfaces

### 5.3.5.1. The *ImgIn* Interface of the eVA IP Core

The image streaming ports *ImgIn* are general purpose image communication interfaces for writing image data from surrounding FPGA logic into the VA IP Core. The *ImgIn* ports consist of a simple FIFO interface plus additional parameter ports. The interface ports are thoroughly parameterized. In addition to the existing parameters, you can define additional registers for forwarding parameters to the connected FPGA logic.

Image data enters the VA IP Core by interface ports of type *ImgIn*. Multiple classes of *ImgIn* ports may be defined and for each of them multiple instances are possible.

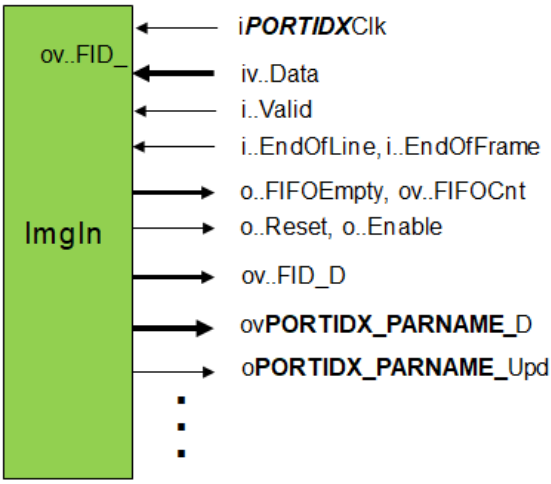



Figure 5.9. Port Layout for Image Input Interface

i = input signal

iv = input values (data)

o = output signal

ov = output values (data)

The following table describes the interface signals where **PORTID** is the name of the corresponding image input port class and **X** is a port number for differentiating several ports of the same class:

| Port | Direction | Width | Description |
|---|---|---|---|
| **iPORTIDXClk** | In | 1 | Clock for writing to FIFO. This input is ignored when the port is configured for synchronous communication. *ImgIn* Interface configuration is described Section 5.3.5.3, 'Defining *ImgIn* Interface Classes'. |
| iv**PORTIDX**Data | In | **PORTID**Width | Write data (interpreted as pixel data, or as and-of-line or and-of-frame flag) |
| i**PORTIDX**Valid | In | 1 | Perform write access |
| i**PORTIDX**EndOfLine | In | 1 | Signal current write access as end-of-line notification. Write data is then not interpreted as pixel data. |
| i**PORTIDX**EndOfFrame | In | 1 | Signals end of frame. If this flag is activated data doesn't contain pixel values. The end-of-frame signal must coincide with the end-of-line signal. |

| Port | Direction | Width | Description |
|---|---|---|---|
| o**PORTIDX**FIFOFull | Out | 1 | Input FIFO is full, no further data is accepted. |
| ov**PORTIDX**FIFOCnt | Out | Ceil Log2(**PORTID**FIFODepth). Ceil Log2 is the number of bits required for representing the value. | Number of words in input FIFO. This signal can be used to generate FIFO flags like *Almost Full*. |
| o**PORTIDX**Reset | Out | 1 | Reset signal of the process |
| o**PORTIDX**Enable | Out | 1 | Enable signal of the process |
| ov**PORTIDX_PARNAME**_D | Out | S | Data of the parameter **PARNAME**. S depends on the selected bit width. This signal is generated for each parameter defined. |
| o**PORTIDX_PARNAME**_Upd | Out | 1 | This signal is set to '1' for one clock cycle when the parameter **PARNAME** is updated from the runtime software. This signal is generated for each parameter defined. |
| ov**PORTIDX**_FID_D | Out | Ceil Log2(N). Ceil Log2 (N) is the number of bits required for representing the value (N-1). | Predefined parameter which notifies about the current image data format. N is the number of image formats specified for this port. |

The following figure shows the timing of image communication through an *ImgIn* interface for a simple example where a frame with two lines of three 32-bit gray pixel values is transferred. In the waveform the name part **PORTIDX** has been substituted by **imgin**.

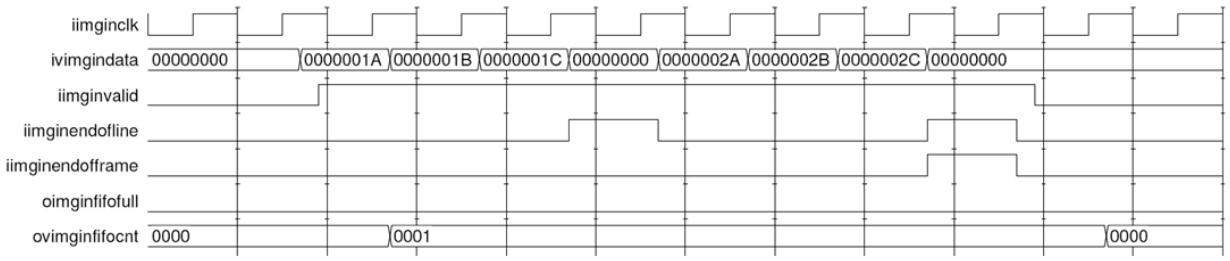



Figure 5.10. Waveform Illustrating the Protocol on an Image Input Port

## 5.3.5.2. Supported ImageIn Formats

Various different protocols can be driven through a single *ImgIn* port. VisualApplets supports the following image formats:

- gray**X**xP: gray image with **X** bits per pixel and parallelism **P**
- rgbYx**P**: color image with **Y**/3 bits per color component (red, green, blue) and parallelism **P**

- hsi**Y**x**P**: color image with **Y**/3 bits per color component (HSI color model) and parallelism **P**
- hsl**Y**x**P**: color image with **Y**/3 bits per color component (HSL color model) and parallelism **P**
- hsv**Y**x**P**: color image with **Y**/3 bits per color component (HSV color model) and parallelism **P**
- yuv**Y**x**P**: color image with **Y**/3 bits per color component (YUV color model) and parallelism **P**
- ycrcb**Y**x**P**: color image with **Y**/3 bits per color component (YCrCb color model) and parallelism **P**
- lab**Y**x**P**: color image with **Y**/3 bits per color component (LAB color model) and parallelism **P**
- xyz**Y**x**P**: color image with **Y**/3 bits per color component (XYZ color model) and parallelism **P**

You can define an *ImgIn* interface class to support any combination of these formats, provided the platform design which is surrounding the VA IP Core supports the appropriate configuration of image communication channels.

You can specify if pixel components are being interpreted as signed or unsigned.

### 5.3.5.3. Defining *ImgIn* Interface Classes

You can define various classes of *ImgIn* interfaces. Those interface classes may differ in number and kind of supported image protocols, supported data width, used clock signals, and in many other aspects. Each *ImgIn* interface class can be available more than one time on the VA IP Core.

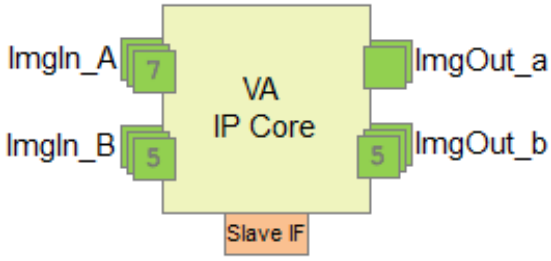



Figure 5.11. Example of eVA IP Core

The example IP Core in the figure above shows two *ImgIn* classes and two *ImgOut* classes. Multiple instances of both classes are available: This VA IP Core provides 7 *ImgIn* ports of *ImgIn* class *ImgIn_A*, 5 *ImgIn* ports of *ImgIn* class *ImgIn_B*, 2 *ImgOut* ports of *ImgOut* class *ImgOut_a*, and 5 *ImgOut* ports of *ImgOut* class *ImgOut_b*.



**Related Operators in VisualApplets**

For each defined *ImgIn* class, one or more platform-specific VisualApplets *ImgIn* operators may be generated later (see Section 5.3.9, 'Defining Hardware-Specific Operators' for details). Those operators can (after IP core integration) be instantiated within VisualApplets and then implement the interface.

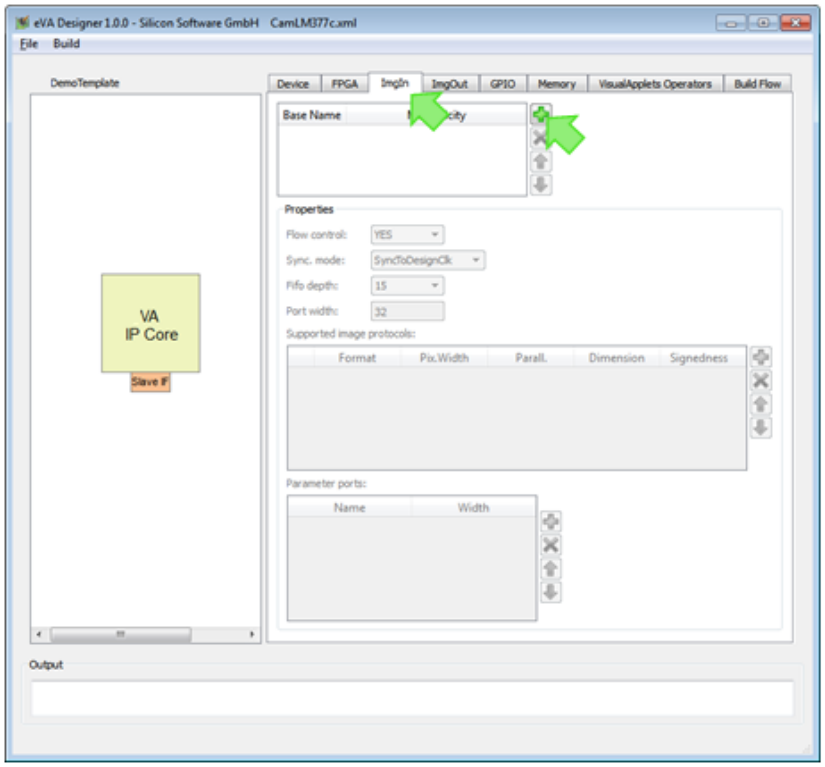
If more than one interface port of the same configuration (class) exists, the instances of the corresponding operator can connect to any of them controlled by the resource management of VisualApplets.

Resource management of VisualApplets: In VisualApplets, for each *ImgIn* class a resource with the same name is set up. In this resources dialog of VisualApplets, the allocation can be defined.

#### 5.3.5.3.1. Setting up *ImgIn* Ports

To define the ports that allow image streaming into the eVA IP Core:

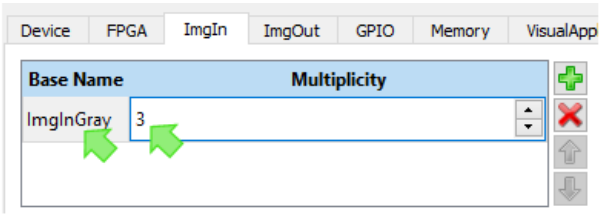
1. Go to the **ImgIn** tab.



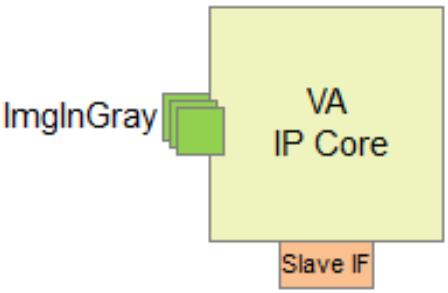

2. Click the plus  icon.

3. Into field **Base Name**, enter the name of the *ImgIn* interface class you want to define. Double-click into the field to write.

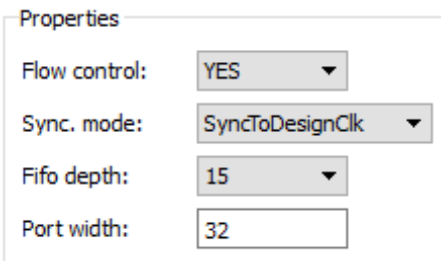
4. In field **Multiplicity**, define how many *ImgIn* interfaces of this class you want to have on your VA IP Core. Double-click into the field to write.




The defined *ImgIn* ports (in our example, 3 interfaces of class *ImgInGray*) are immediately visible in the graphical representation of the IP Core:




5. Define further properties of the *ImgIn* interface class:




**Flow control:** This parameter decides if the VisualApplets application is allowed to temporarily stop the incoming image data flow or not. If set to YES, the internal flow control mechanism for pausing input data will be implemented. If set to NO, the flow control mechanism will not be implemented. Implementation of the flow control mechanism is sensible if you have a stoppable image source connected to the *ImgIn* port. If you use a non-stoppable source, select NO.

- **Yes**: flow control mechanism for pausing input data is implemented.
- **NO**: no flow control mechanism for pausing input data is implemented.

**Sync. mode:** This parameter signals the relation of the image interface clock (*iPORTIDXClk* on the *ImgIn* port) and the design clock (*iDesignClk* on the IP Core). Following values are possible:

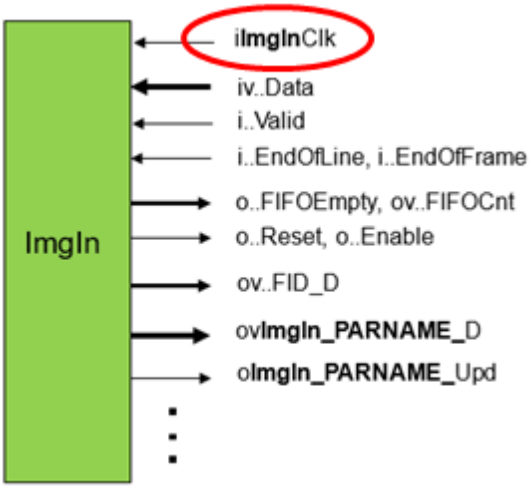
- *SyncToDesignClk* – interface ports are synchronous to *iDesignClk*. The external clock input of the image interface port is ignored.
- *SyncToDesignClk2x* – interface ports are synchronous to *iDesignClk2x*. The external clock input of the *ImgIn* port is ignored.



**Double Pixel Depth Available**

If you use *iDesignClk2x* for the *ImgIn* port, you may define an image protocol with twice the bit depth you define for port *iv..Data* (i.e., the product of bit width and parallelism can be double-size). In this case, an automatic parallel-up with factor 2 is carried out by the system.

- **Async** – asynchronous interface: The interface ports are synchronized to the external clock input of the *ImgIn* interface.




If you select **Async**, you need to

- provide a clock signal at the **iImgInClk** port of the *ImgIn* interface, and to

- define clock domain transition constraints in the synthesis constraints (constraints file).



**Double Pixel Depth Available**

If you provide a clock signal frequency at the *iImgInClk* port that is much higher than *iDesignClk*, you may define an image protocol with twice the bit depth you define for port *iv..Data* (i.e., the product of bit width and parallelism can be double-size). In this case, an automatic parallel-up with factor 2 is carried out by the system.



**Use *iDesignClk* or *iDesignClk2x***

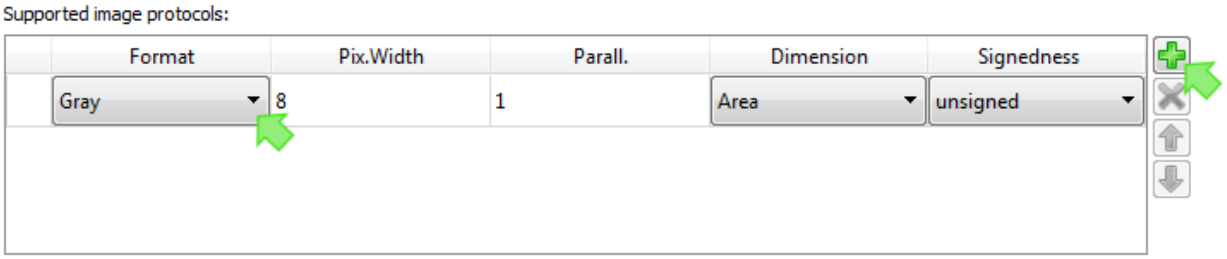
Basler recommends to use *iDesignClk* or *iDesignClk2x* for synchronization whenever possible.

**Fifo depth:** Define here the depth of the buffer FIFO for input data which at least needs to be implemented by an attached VisualApplets interface operator. The value must be a power of two minus 1 between 15 and 1023.

**Port width:** Define here the width of the image data port.

### 5.3.5.3.2. Defining Image Protocols for an *ImgIn* Interface Class

You need to define the image protocols you want this *ImgIn* interface class to support. Define one protocol after the other until all protocols you want to specify are there.

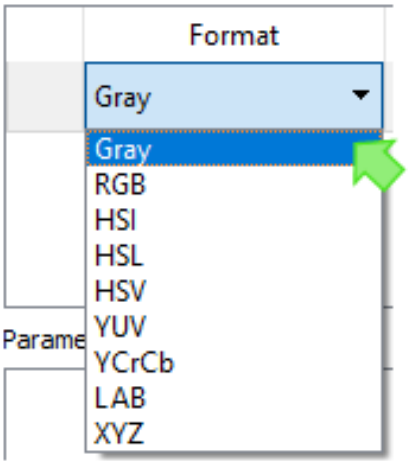




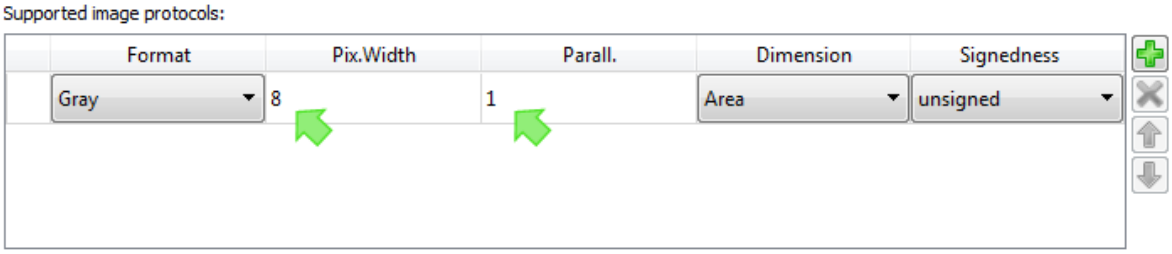

**Related Operators in VisualApplets**

After integration of the VA IP Core into your FPGA design, the developer of the image processing application instantiates an *ImgIn* operator in VisualApplets and defines which of the protocols supported by the *ImgIn* interface (and the operator itself) is implemented by this operator instance in a given application.

1. Click on the arrow in column **Format** and select the format of your choice from the format list:




Find more detailed description on the supported image formats in Section 5.3.5.2, 'Supported ImageIn Formats'.




2. Double-click on the field in column **Pix.Width**.

   Enter the required pixel data bit width (value range: [1..64]).

   The pixel data width is limited to 64 bit. The pixel data width for all non-gray color formats must be a multiple of 3 and is limited to 63 bit.

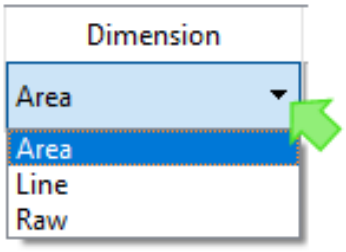
3. Double-click on the field in column **Parall**. Enter the required parallelism.

   The parallelism P defines the number of pixels which are contained in a single data word at the interface port. It must be chosen from the following set of allowed values: **P** = {1, 2, 4, 8, 16, 32, 64}. Packing of image data into words of a given interface width **N** (**Port width**, see above) must follow certain rules:

   - The data of all **P** pixels must fit in a single word of length **N**. The data is stored LSB aligned which means that for a pixel width Z (**Z**=**X** for grey, Z=**Y** for color) data is distributed as follows: Pixel[0]->Bits[0..**Z**-1] .. Pixel[**P**-1]->Bits[(**P**-1)*Z..**P***Z-1].
   - For RGB images the three color components are packed LSB aligned into a sub word [0..**Y**-1] in the following order: red uses the bits [0..**Y**/3-1 ], green the bits [**Y**/3..2***Y**/3-1] and blue the bits [2***Y**/3..3***Y**/3-1].
   - For HSI color images the same rules than for RGB applies where H takes the role of red, S that of green and I the role of blue.
   - For HSL color images the same rules than for RGB applies where H takes the role of red, S that of green and L the role of blue.
   - For HSV color images the same rules than for RGB applies where H takes the role of red, S that of green and V the role of blue.
   - For YUV color images the same rules than for RGB applies where Y takes the role of red, U that of green and V the role of blue.

- For YCrCb color images the same rules than for RGB applies where Y takes the role of red, Cr that of green and Cb the role of blue.
- For LAB color images the same rules than for RGB applies where L takes the role of red, A that of green and B the role of blue.
- For XYZ color images the same rules than for RGB applies where X takes the role of red, Y that of green and Z the role of blue.

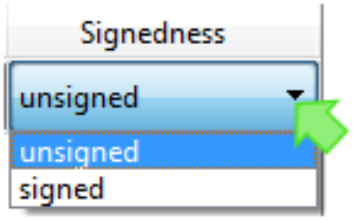
4. Click on the arrow in column **Dimension** to select the image dimension of the protocol you are defining.




The meaning of the dimension is as follows:

- **Area** (default): The image is structured by end-of-line and end-of-frame markers. In VisualApplets, image dimension **Area** is named *2D*. An area camera could be an 2D image source.
- **Line**: The image is structured by end-of-line markers. There are no end-of-frame markers which divide the incoming lines into frames. In VisualApplets, image dimension **Line** is named *1D*. A line camera could be an 1D image source.
- **Raw**: There are no end-of-line and no end-of-frame markers which divide the incoming image data into lines and frames. The image-in stream consists of an endless pixel stream with a width of 1 pixel. In VisualApplets, image dimension **Raw** is named *0D*.

5. Click on the arrow in column **Signedness** and select the image dimension of the protocol you are defining.




- Unsigned (default): Pixel data are interpreted as unsigned pixel components.
- Signed: Pixel data are interpreted as signed pixel components

6. Repeat steps 1 - 5 until you have defined all image protocols you want the image class to support.



**Internal ID for Image Protocol Definitions**

Implicitly it is assumed that the kernel size is 1x1. Each listed protocol is numbered, the list starts from zero. These numbers form the protocol IDs for a given *ImgIn* class. The ID is not displayed on the GUI. Thus, each *ImgIn* class you define has an internal ID list of the image protocols you define.

In one of the next steps, you can define one or more VisualApplets *ImgIn* operators the instances of which can connect to the defined *ImgIn* class (details how to define operators you find in Section 5.3.9, 'Defining Hardware-Specific Operators'). To each of

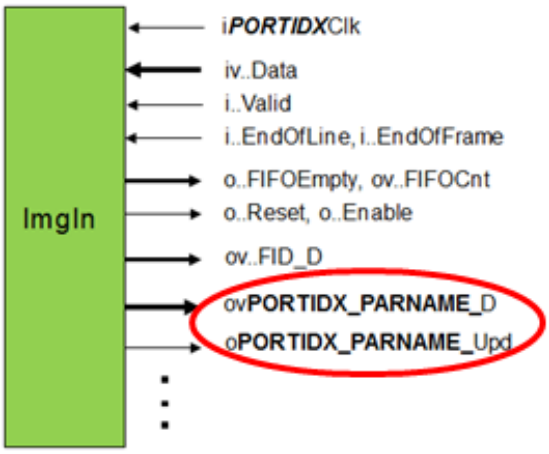
these operators you can assign all or a subset of the image protocols you specified for the concerning image port.

In VisualApplets, the *ImgIn* operator will provide a parameter where the VisualApplets user can select an image format from a list of options. According to the selection, the corresponding ID will be assigned to the related IP core *ImgIn* port (ov..FID_D). This enables the attached glue logic to adapt its behavior according to the selected format.

### 5.3.5.3.3. Defining Additional Parameters for an *ImgIn* Interface Class

You can define additional parameters you want your *ImgIn* interface class to have. These (dynamic) parameters can be set during runtime. They can be accessed via the runtime interface (see Section 5.5, 'Runtime Software Interface').

These parameters are communicated via the register slave interface (see Section 5.2.2, 'Register Slave Interface') of the VA IP Core.

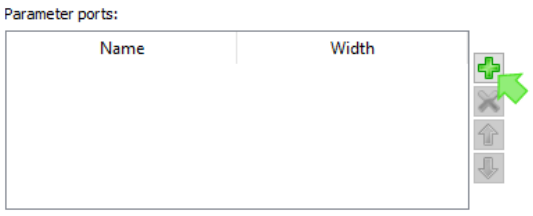



ov**PORTIDX_PARNAME**_D: Data of the parameter **PARNAME**. Direction: Out. You can define the bit width. This signal is generated for each parameter defined.

o**PORTIDX_PARNAME**_Upd: Direction: Out. Bit width: 1. This signal is set to '1' for one clock cycle when the parameter **PARNAME** is updated from the runtime software. This signal is generated for each parameter defined.

To define additional parameters for the *ImgIn* interface class:

1. Go to the **Parameter ports** area of the program window.




2. Click the plus  icon.

3. Into field **Name**, enter the name of the parameter you want to define. Double-click into the field to write.

4. In field **Width**, define the width of the parameter port {1,64}. Double-click into the field to write.




**Mapping between Slave Interface Data Width and Actual Register Width**

Registers which are accessed through the register interface may have any width between 1 and 64. Mapping between the slave interface data width and the actual register width of a VisualApplets parameter is done automatically. When the width of a register is bigger than the width of the register interface the runtime software will divide the access automatically. A single parameter then consumes more than one register address.

5. Repeat steps 2 – 4 until you have defined all parameters you want the image class to support.

   After carrying out all steps described in Section 5.3.5.3.1, '**Setting up *ImgIn* Ports**', Section 5.3.5.3.2, '**Defining Image Protocols for an *ImgIn* Interface Class**', and Section 5.3.5.3.3, '**Defining Additional Parameters for an *ImgIn* Interface Class**', you have set up your first *ImgIn* interface class.

6. If you need more *ImgIn* interface classes, define the next *ImgIn* Interface class by re-starting with Section 5.3.6.3.1, '**Setting up *ImgOut* Ports**' again.

**Define as Many *ImgIn* Interface Classes as You Need**

You can define as many interface classes as you need. For setting up the next *ImgIn* interface class, start with the steps described in section Section 5.3.5.3.1, '**Setting up *ImgIn* Ports**' again.

## 5.3.6. Entering Descriptions of Required *ImgOut* Interfaces

### 5.3.6.1. The *ImgOut* Interface of the VA IP Core

The image streaming ports *ImgOut* are general purpose image communication interfaces for writing image data from the VA IP Core into the surrounding FPGA logic. The *ImgOut* ports consist of a simple FIFO interface plus additional parameter ports. The interface ports are thoroughly parameterized. In addition to the existing parameters, you can define additional registers for forwarding parameters to the connected FPGA logic.

Image data leaves the VA IP Core by interface ports of type *ImgOut*. Multiple classes of *ImgOut* ports may be defined and for each of them multiple instances are possible.

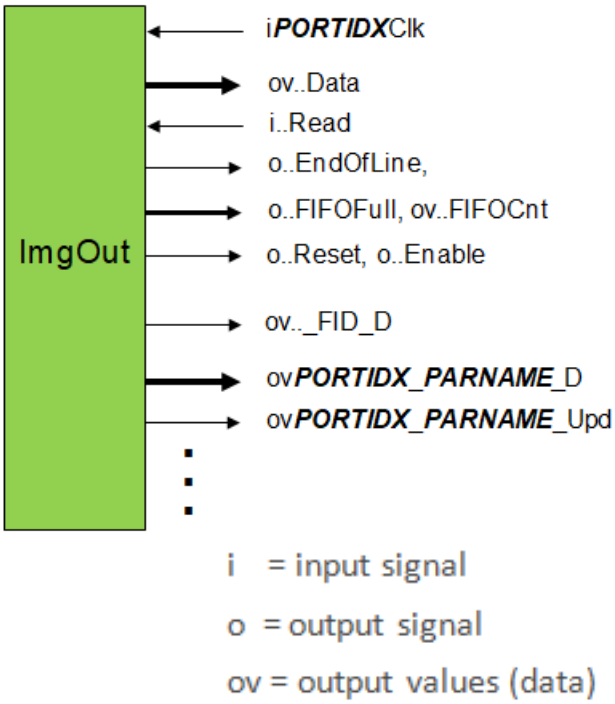



Figure 5.12. Port Layout for Image Output Interface

The following table describes the interface signals where **PORTID** is the name of the corresponding image input port class and **X** is a port number for differentiating several ports of the same class:

| Port | Direction | Width | Description |
|---|---|---|---|
| i**PORTIDX**Clk | In | 1 | Clock for reading from FIFO. This input is ignored when the port is configured for synchronous communication. (*ImgOut* interface configuration is described in Section 5.3.6.3, 'Defining *ImgOut* Interface Classes'.) |
| iv**PORTIDX**Data | Out | **PORTID**Width | Data for custom read |
| i**PORTIDX**Read | In | 1 | Perform read access |
| i**PORTIDX**EndOfLine | Out | 1 | Signals end of line. If this flag is activated data doesn't contain pixel values. |
| i**PORTIDX**EndOfFrame | Out | 1 | Signals end of frame. If this flag is activated data doesn't contain pixel values. The end-of-frame signal must coincide with the end-of-line signal. |
| o**PORTIDX**FIFOEmpty | Out | 1 | Output FIFO is empty |
| ov**PORTIDX**FIFOCnt | Out | Ceil Log2( **PORTID**FIFODepth | Number of words in output FIFO. This signal |

| Port | Direction | Width | Description |
|---|---|---|---|
| | | ) | can be used to generate FIFO flags like "Almost Empty". |
| o**PORTIDX**Reset | Out | 1 | Reset signal of the process |
| o**PORTIDX**Enable | Out | 1 | Enable signal of the process |
| ov**PORTIDX_PARNAME**_D | Out | S | Data of the parameter **PARNAME**. S depends on the selected bit width. This signal is generated for each parameter defined. |
| o**PORTIDX_PARNAME**_Upd | Out | 1 | This signal is set to '1' for one clock cycle when the parameter **PARNAME** is updated from the runtime software. This signal is generated for each parameter defined. |
| ov**PORTIDX**_FID_D | Out | Ceil Log2(N). Ceil Log2 (N) is the number of bits required for representing the value (N-1). | Predefined parameter which notifies about the current image data format. N is the number of image protocols specified for this port. |

The following figure shows the timing of image communication through an *ImgOut* interface for a simple example: A frame with two lines of three 32-bit gray pixel values is transferred. In the waveform the name part **PORTIDX** has been substituted by *imgout*.

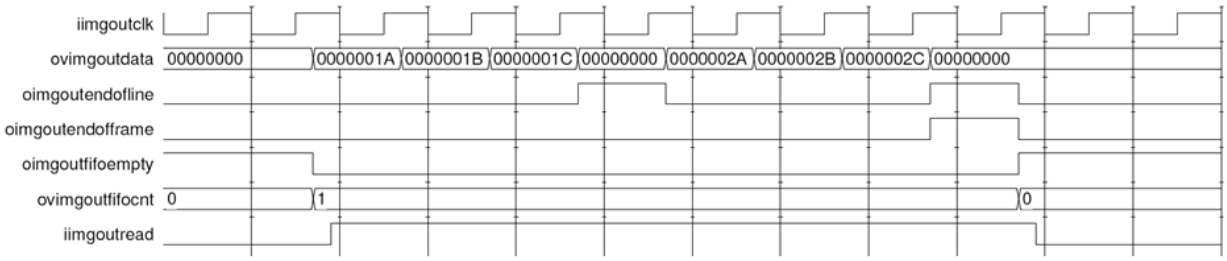



Figure 5.13. Waveform Illustrating the Protocol on an Image Output Port

### 5.3.6.2. Supported *ImageOut* Formats

Various different protocols can be driven through a single *ImgOut* port. VisualApplets supports the following image formats:

- gray**X**xP: gray image with **X** bits per pixel and parallelism **P**
- rgbYx**P**: color image with **Y**/3 bits per color component (red, green, blue) and parallelism **P**
- hsi**Y**x**P**: color image with **Y**/3 bits per color component (HSI color model) and parallelism **P**
- hsl**Y**x**P**: color image with **Y**/3 bits per color component (HSL color model) and parallelism **P**
- hsv**Y**x**P**: color image with **Y**/3 bits per color component (HSV color model) and parallelism **P**

- yuv**Y**x**P**: color image with **Y**/3 bits per color component (YUV color model) and parallelism **P**
- ycrcb**Y**x**P**: color image with **Y**/3 bits per color component (YCrCb color model) and parallelism **P**
- lab**Y**x**P**: color image with **Y**/3 bits per color component (LAB color model) and parallelism **P**
- xyz**Y**x**P**: color image with **Y**/3 bits per color component (XYZ color model) and parallelism **P**

You can define an *ImgOut* interface class to support any combination of these formats, provided the platform design which is surrounding the VA IP Core supports the appropriate configuration of image communication channels.

### 5.3.6.3. Defining *ImgOut* Interface Classes

You can define various classes of *ImgOut* interfaces. Those interface classes may differ in number and kind of supported image protocols, supported data width, used clock signals, and in many other aspects. Each *ImgOut* interface class can be available more than one time on the VA IP Core.

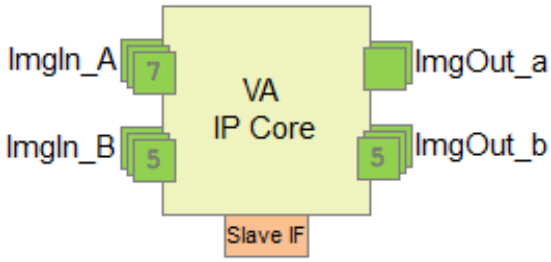



Figure 5.14. Example of eVA IP Core

The example in the figure above shows two *ImgIn* classes and two *ImgOut* classes. Multiple instances of both classes are available: This VA IP Core provides 7 *ImgIn* ports of *ImgIn* class *ImgIn_A*, 5 *ImgIn* ports of *ImgIn* class *ImgIn_B*, 2 *ImgOut* ports of *ImgOut* class *ImgOut_a*, and 5 *ImgOut* ports of *ImgOut* class *ImgOut_b*.



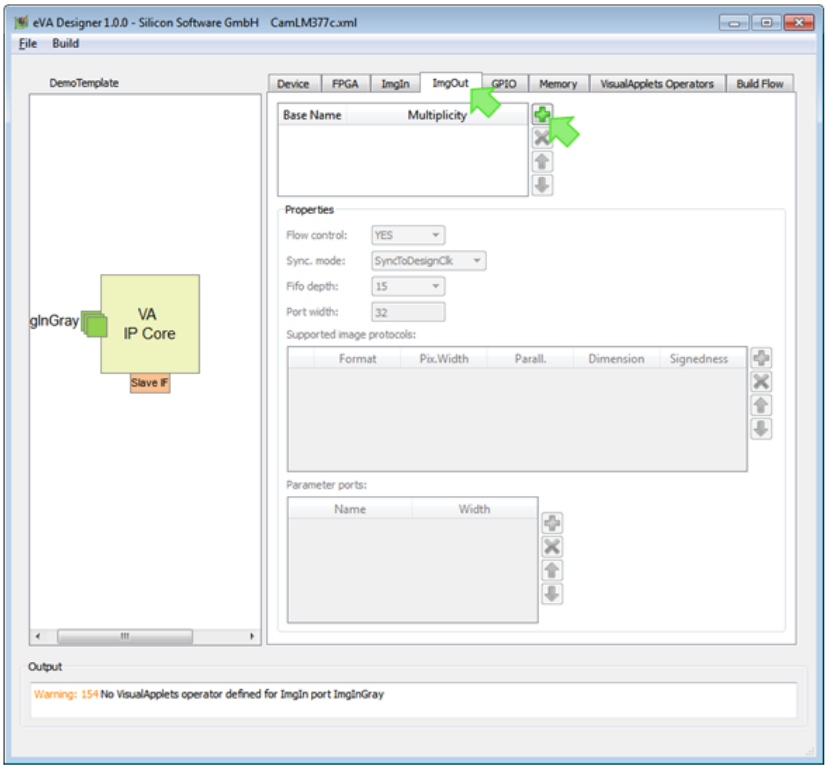
**Related Operators in VisualApplets**

For each defined ImgOut class, you can later define one or more platform-specific VisualApplets ImgOut operators (see Section 5.3.9, 'Defining Hardware-Specific Operators' for details). Those operators can (after IP core integration) be instantiated within VisualApplets and then implement the interface. If more than one interface port of the same configuration (class) exists, the instances of the corresponding operator can connect to any of them controlled by the resource management of VisualApplets.

#### 5.3.6.3.1. Setting up *ImgOut* Ports

To define the ports that allow image streaming from the VA IP Core to the surrounding FPGA design:

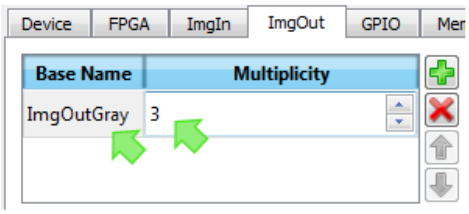
1. Go to the **ImgOut** tab.



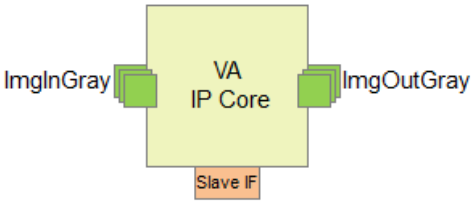

2. Click the plus  icon.

3. Into field **Base Name**, enter the name of the *ImgOut* interface class you want to define. Double-click into the field to write.

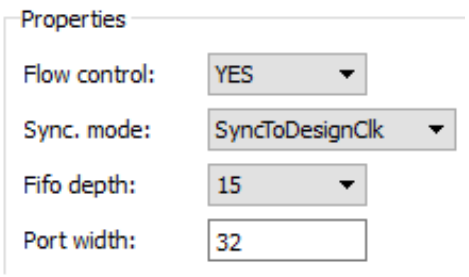
4. In field **Multiplicity**, define how many *ImgOut* interfaces of this class you want to have on your VA IP Core. Double-click into the field to write.




The defined *ImgOut* ports (in our example, 3 interfaces of class *ImgOutGray*) are immediately visible in the graphical representation of the IP Core:




5. Define further properties of the *ImgOut* interface class:




**Flow control {Yes,No}:**

- **Yes:** If you select Yes, a flow control mechanism for pausing output data is implemented. Value Yes allows VisualApplets designs to block reading from the output FIFO for any duration. A FIFO-full signal signals that no further data may be written.
- **NO:** If you select No, no flow control mechanism for blocking output data is implemented.

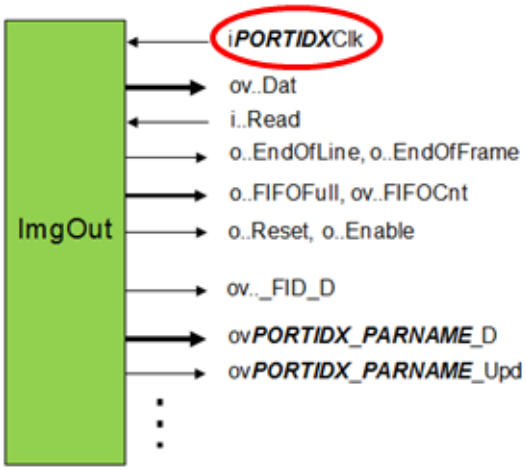
**Sync. mode:** This parameter signals the relation of the image interface clock (*iPORTIDXClk* on the *ImgOut* port) and the design clock (*iDesignClk* on the IP Core). Following values are possible:

- *SyncToDesignClk* – interface ports are synchronous to *iDesignClk*. The external clock input on the *ImgOut* port is ignored.
- *SyncToDesignClk2x* – interface ports are synchronous to *iDesignClk2x*. The external clock input on the *ImgOut* port is ignored.

**Double Pixel Depth Available**

If you use *iDesignClk2x* for the *ImgOut* port, you may define an image protocol with twice the bit depth you define for port *ov..Data* (i.e., the product of bit width and parallelism can be double-size). In this case, an automatic parallel-up with factor 2 is carried out by the system.

- **Async** – asynchronous interface: The interface ports are synchronized to the external clock input of the *ImgOut* interface.




If you select **Async**, you need to

- provide a clock signal at the **iPortIDXClk** port of the *ImgOut* interface, and to
- define clock domain transition constraints in the synthesis constraints (constraints file).

**Double Pixel Depth Available**

If you use *iDesignClk2x* for the *ImgOut* port, you may define an image protocol with twice the bit depth you define for port ov..Data (i.e., the product of bit width and parallelism can be double-size). In this case, an automatic parallel-up with factor 2 is carried out by the system.

**Use *iDesignClk* or *iDesignClk2x***

Basler recommends to use *iDesignClk* or *iDesignClk2x* for synchronization whenever possible.

**Fifo depth:** Define here depth of the buffer FIFO for output data which at least needs to be implemented by an attached Visual Applets interface operator. The value must be a power of two minus 1 between 15 and 1023.

**Port width:** Define here the width of the image data port.

### 5.3.6.3.2. Defining Image Protocols for an *ImgOut* Interface Class

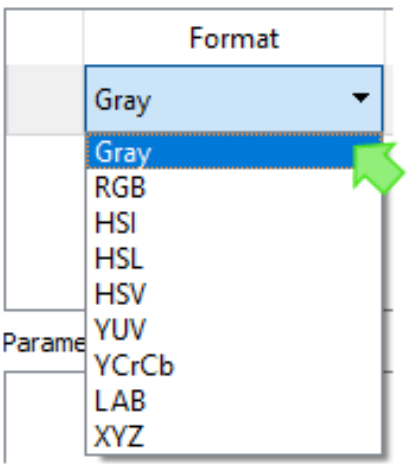
You need to define the image protocols you want this *ImgOut* interface class to support. You define one protocol after the other until all protocols you want to specify are there.



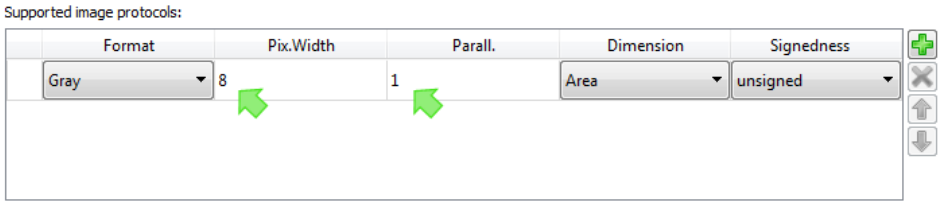

**Related Operators in VisualApplets**

After integration of the VA IP Core into your FPGA design, the developer of the image processing application instantiates an *ImgOut* operator in VisualApplets and defines which of the protocols supported by the *ImgOut* interface (and the operator itself) is implemented by this operator instance in a given application.

1. Click on the arrow in column **Format** and select the format of your choice from the format list:



A more detailed description of the supported image formats you find in Section 5.3.6.2, 'Supported *ImageOut* Formats'.




2. Double-click on the field in column **Pix.Width**.

   Enter the required pixel data bit width (value range: [1..64]).

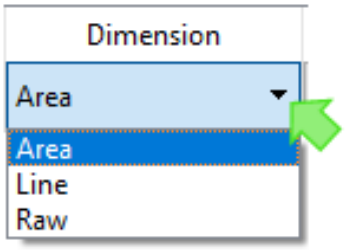
   The pixel data width is limited to 64 bit. The pixel data width for all non-gray color formats must be a multiple of 3 and is limited to 63 bit.

3. Double-click on the field in column **Parall**. Enter the required parallelism.

   The parallelism P defines the number of pixels which are contained in a single data word at the interface port. It must be chosen from the following set of allowed values: **P** = {1, 2, 4, 8, 16, 32, 64}. Packing of image data into words of a given interface width **N** (**Port width**, see above) must follow certain rules:

   - The data of all **P** pixels must fit in a single word of length **N**. The data is stored LSB aligned which means that for a pixel width Z (**Z**=**X** for grey, Z=**Y** for color) data is distributed as follows: Pixel[0]->Bits[0..**Z**-1] .. Pixel[**P**-1]->Bits[(**P**-1)*Z..**P***Z-1].
   - For RGB images the three color components are packed LSB aligned into a sub word [0..**Y**-1] in the following order: red uses the bits [0..**Y**/3-1 ], green the bits [**Y**/3..2*Y/3-1] and blue the bits [2*Y/3..3*Y/3-1].
   - For HSI color images the same rules than for RGB applies where H takes the role of red, S that of green and I the role of blue.
   - For HSL color images the same rules than for RGB applies where H takes the role of red, S that of green and L the role of blue.
   - For HSV color images the same rules than for RGB applies where H takes the role of red, S that of green and V the role of blue.
   - For YUV color images the same rules than for RGB applies where Y takes the role of red, U that of green and V the role of blue.
   - For YCrCb color images the same rules than for RGB applies where Y takes the role of red, Cr that of green and Cb the role of blue.
   - For LAB color images the same rules than for RGB applies where L takes the role of red, A that of green and B the role of blue.
   - For XYZ color images the same rules than for RGB applies where X takes the role of red, Y that of green and Z the role of blue.

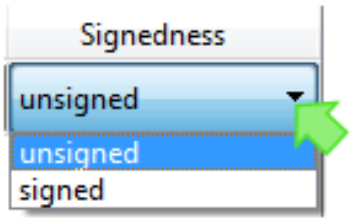
4. Click on the arrow in column **Dimension** to select the image dimension of the protocol you are defining.




The meaning of the dimension is as follows:

- **Area** (default): The image is structured by end-of-line and end-of-frame markers. In VisualApplets, image dimension **Area** is named "2D".
- **Line**: The image is structured by end-of-line markers. There are no end-of-frame markers which divide the incoming lines into frames. In VisualApplets, image dimension **Line** is named "1D".
- **Raw**: There are no end-of-line and no end-of-frame markers which divide the incoming image data into lines and frames. The image-in stream consists of an endless pixel stream with a width of 1 pixel. In VisualApplets, image dimension **Raw** is named "0D". A line camera could be an 0-D image source.

5. Click on the arrow in column **Signedness** and select the image dimension of the protocol you are defining.




- Unsigned (default): Pixel data are packed as unsigned pixel components.
- Signed: Pixel data are packed as signed pixel components

6. Repeat steps 1 - 5 until you have defined all image protocols you want the image class to support.



**Internal ID for Image Protocol Definitions**

Implicitly it is assumed that the kernel size is 1x1. Each listed protocol is numbered, the list starts from zero. These numbers form the protocol IDs for a given *ImgOut* class. The ID is not displayed on the GUI. Thus, each *ImgOut* class you define has an internal ID list of the image protocols you define.

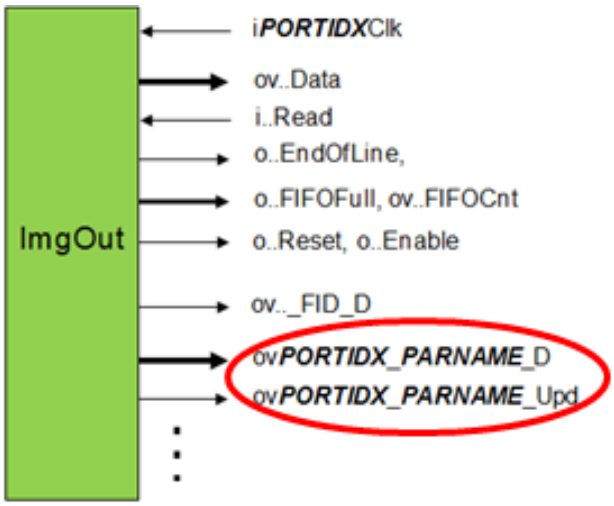
In one of the next steps, you can define VisualApplets *ImgOut* operators. The instances of these operators can connect to the defined *ImgOut* class (details how to define operators you find in Section 5.3.9, 'Defining Hardware-Specific Operators'). To each *ImgOut* operator you can assign all or a subset of the image protocols you specified for the concerning image port.

In VisualApplets, the *ImgOut* operator will provide a parameter where the VisualApplets user can select an image format from of a list of options. According to the selection, the corresponding ID will be assigned to the related IP core *ImgOut* interface port (ov..FID_D). This enables the attached glue logic to adapt its behavior according to the selected format.

### 5.3.6.3.3. Defining Additional Parameters for an *ImgOut* Interface Class

You can define additional parameters you want your *ImgOut* interface class to have. These (dynamic) parameters can be set and re-set during runtime. They can be accessed via the runtime interface (see Section 5.5, 'Runtime Software Interface').

During runtime they are communicated via the register slave interface (see Section 5.2.2, 'Register Slave Interface') of the VA IP Core.




ov**PORTIDX_PARNAME**_D: Data of the parameter **PARNAME**. Direction: Out. You can define the bit width as described below. This signal is generated for each parameter defined.

o**PORTIDX_PARNAME**_Upd: Direction: Out. Bit width: 1. This signal is set to '1' for one clock cycle when the parameter **PARNAME** is updated from the runtime software. This signal is generated for each parameter defined.

To define additional parameters for the *ImgOut* interface class:

1. Go to the **Parameter ports** area of the program window.




2. Click the plus  icon.

3. Into field **Name**, enter the name of the parameter you want to define. Double-click into the field to write.

4. In field **Width**, define the width of the parameter port {1,64}. Double-click into the field to write.




**Mapping between Slave Interface Data Width and Actual Register Width**

Registers which are accessed through the register interface may have any width between 1 and 64. Mapping between the slave interface data width and the actual register width of a VisualApplets parameter is done automatically. When the width of a register is bigger than the width of the register interface the runtime software will divide the access automatically. A single parameter then consumes more than one register address.

5. Repeat steps 2 – 4 until you have defined all parameters you want the image class to support.

   After carrying out all steps described in Section 5.3.6.3.1, '**Setting up *ImgOut* Ports**', Section 5.3.6.3.2, '**Defining Image Protocols for an *ImgOut* Interface Class**', and Section 5.3.6.3.3, '**Defining Additional Parameters for an *ImgOut* Interface Class**', you have set up your first *ImgOut* interface class.

6. If you need more *ImgOut* interface classes, define the next *ImgOut* interface class by re-starting with Section 5.3.6.3.1, '**Setting up *ImgOut* Ports**' again.

**Define as Many *ImgOut* Interface Classes as You Need**

You can define as many interface classes as you need. For setting up the next *ImgOut* interface class, start with the steps described in Section 5.3.6.3.1, '**Setting up *ImgOut* Ports**' again.

## 5.3.7. Entering GPIO Definitions

General purpose signals IN (GPIs) and General purpose signals OUT (GPOs) may enter or leave the VA IP Core. These signals can be used for triggering and process control.

The number of GPI and GPO signals you can configure as described below.

Any GPI or GPO signal which has been defined in the eVA platform description has a corresponding input or output port in the VA IP Core. The signal ports will get the following names when **NAME** is the name of the GPI or GPO signal as it will show up in VisualApplets:

- iSig_**NAME** for an input signal of width 1
- oSig_**NAME** for an output signal of width 1

All dedicated I/O signals are synchronous to the clock *iDesignClk*.

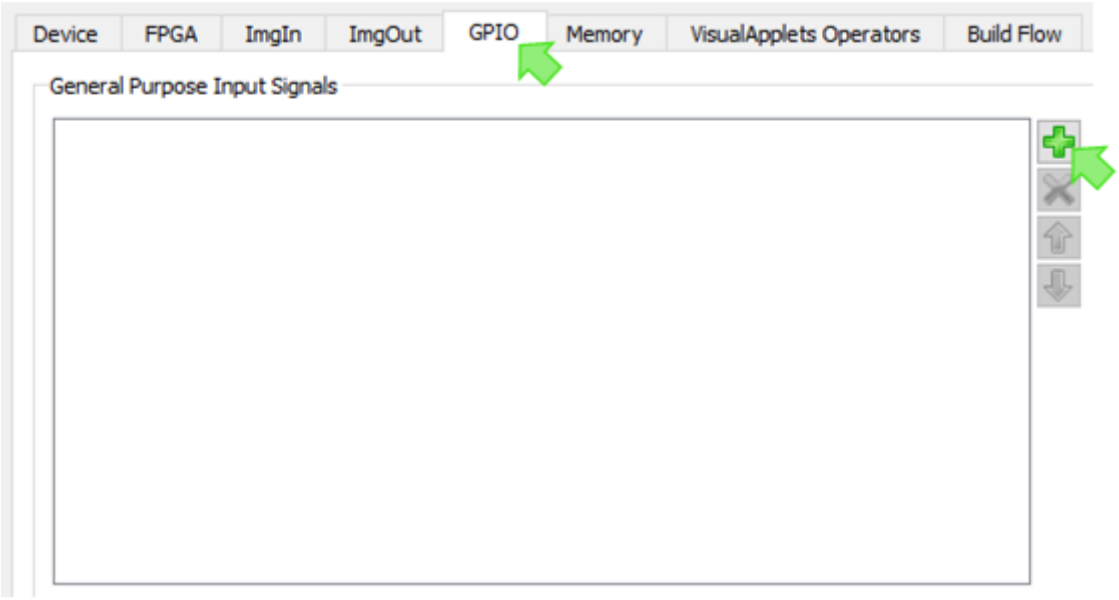



| Port | Direction | Width | Description |
|---|---|---|---|
| iSig_**NAME** | In | 1 | Input signal on GPI port |

| Port | Direction | Width | Description |
|---|---|---|---|
| oSig_**NAME** | Out | 1 | Output signal on GPO port |

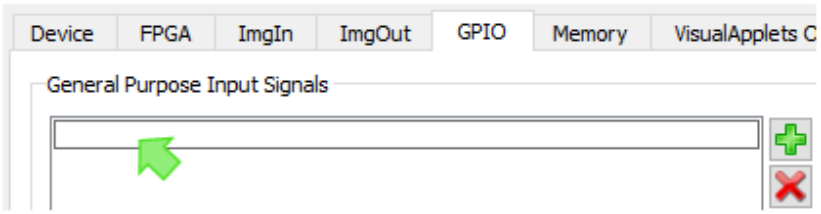
To define the GPI ports of the VA IP Core:

1. Go to the **GPIO** tab.




2. In the **General Purpose Input Signals** area, click the plus  icon.

3. Into the field that is available now, enter the name of the GPI port you want to define. Double-click into the field to write.

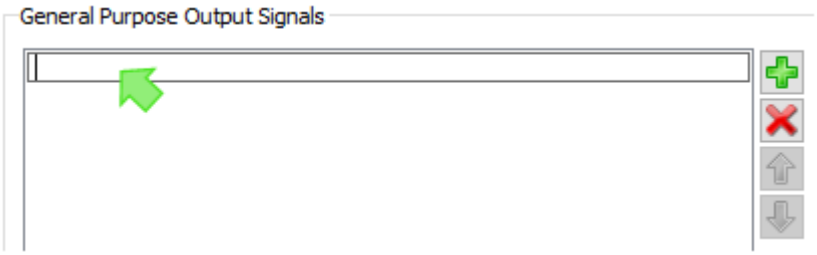



4. Repeat steps 3 and 4 until you have defined all GPI ports you want to have available on your VA IP Core.

5. Go to the **General Purpose Output Signals** area.

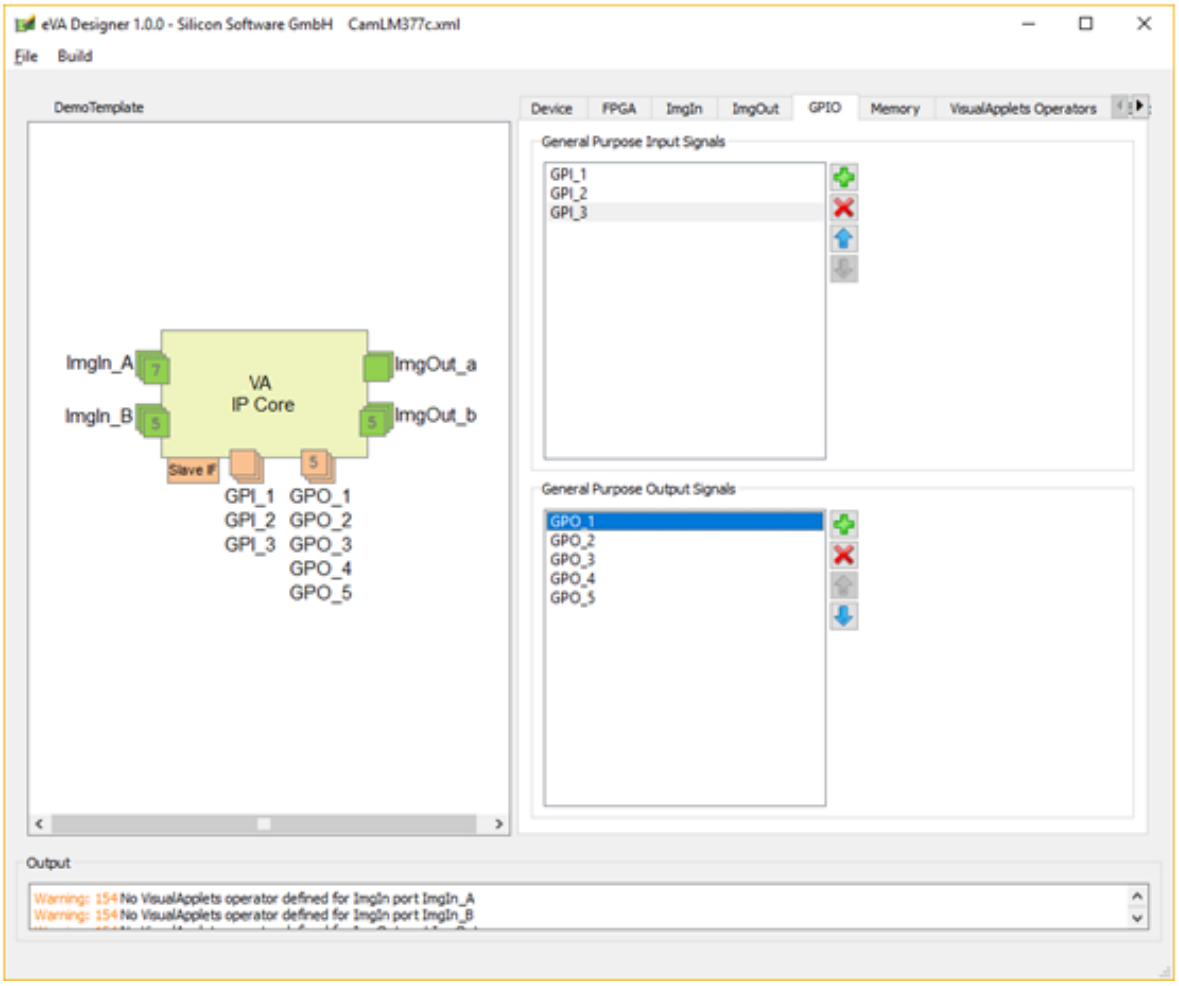
6. Click the plus  icon.

7. Into the field that is available now, enter the name of the GPO port you want to define. Double-click into the field to write.




8. Repeat steps 7 and 8 until you have defined all GPO ports you want to have available on your VA IP Core.

   The defined GPI and GPO ports are immediately visible in the graphical representation of the IP Core:






**Related Operators in VisualApplets**

For the GPI and GPO signals, you can later define one or more platform-specific VisualApplets GPIO operators (see Section 5.3.9, 'Defining Hardware-Specific Operators' for details). These operators are made up by connectors to one or several of the GPIO resources.

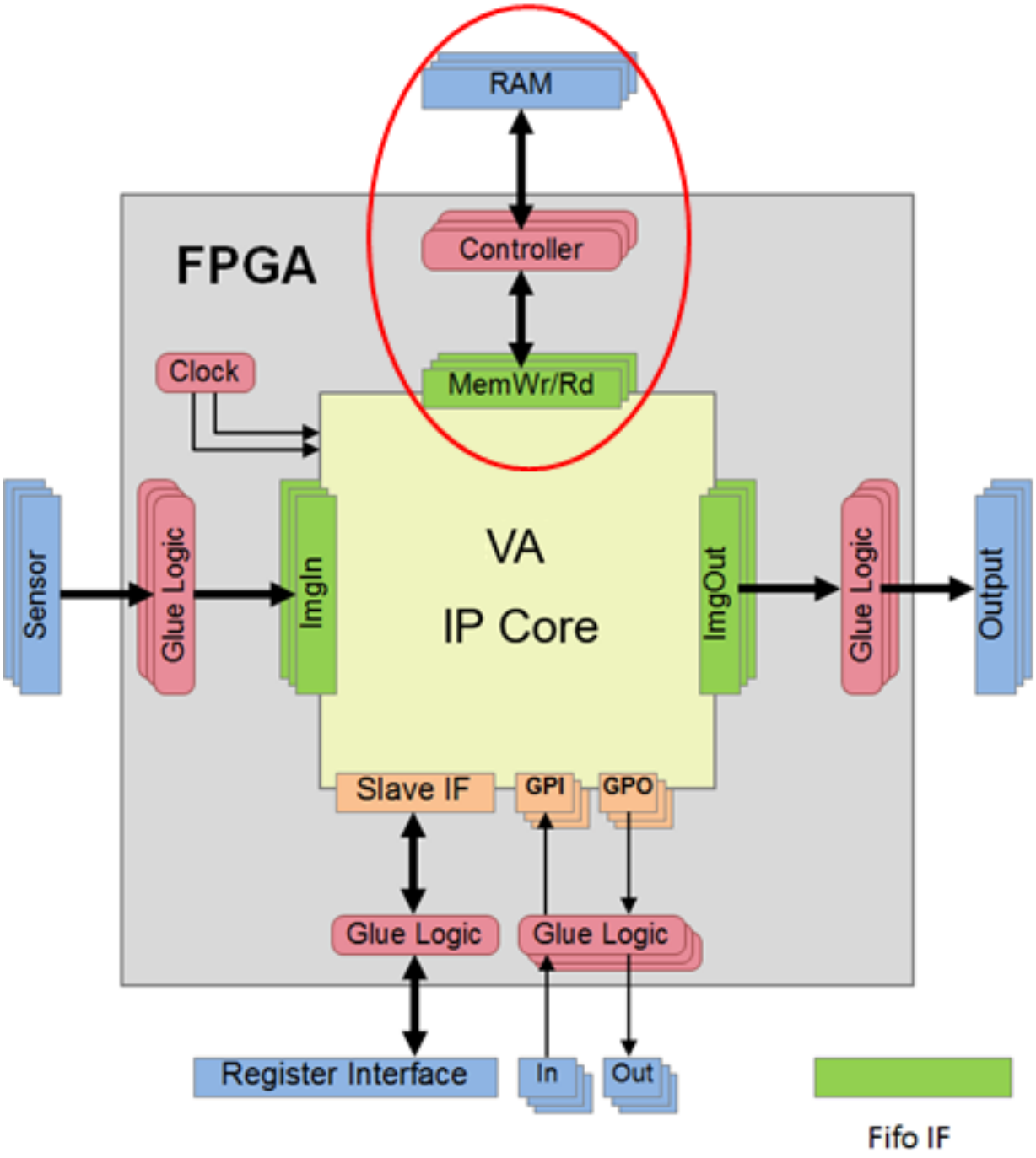
These operators can (after IP core integration) be instantiated within VisualApplets and then implement access to one or more GPI and/or GPO ports of the IP Core.

The VisualApplets environment checks for resource conflicts (the same input/output pin cannot be used by two operator instances).

### 5.3.8. Defining Required Memory Interfaces

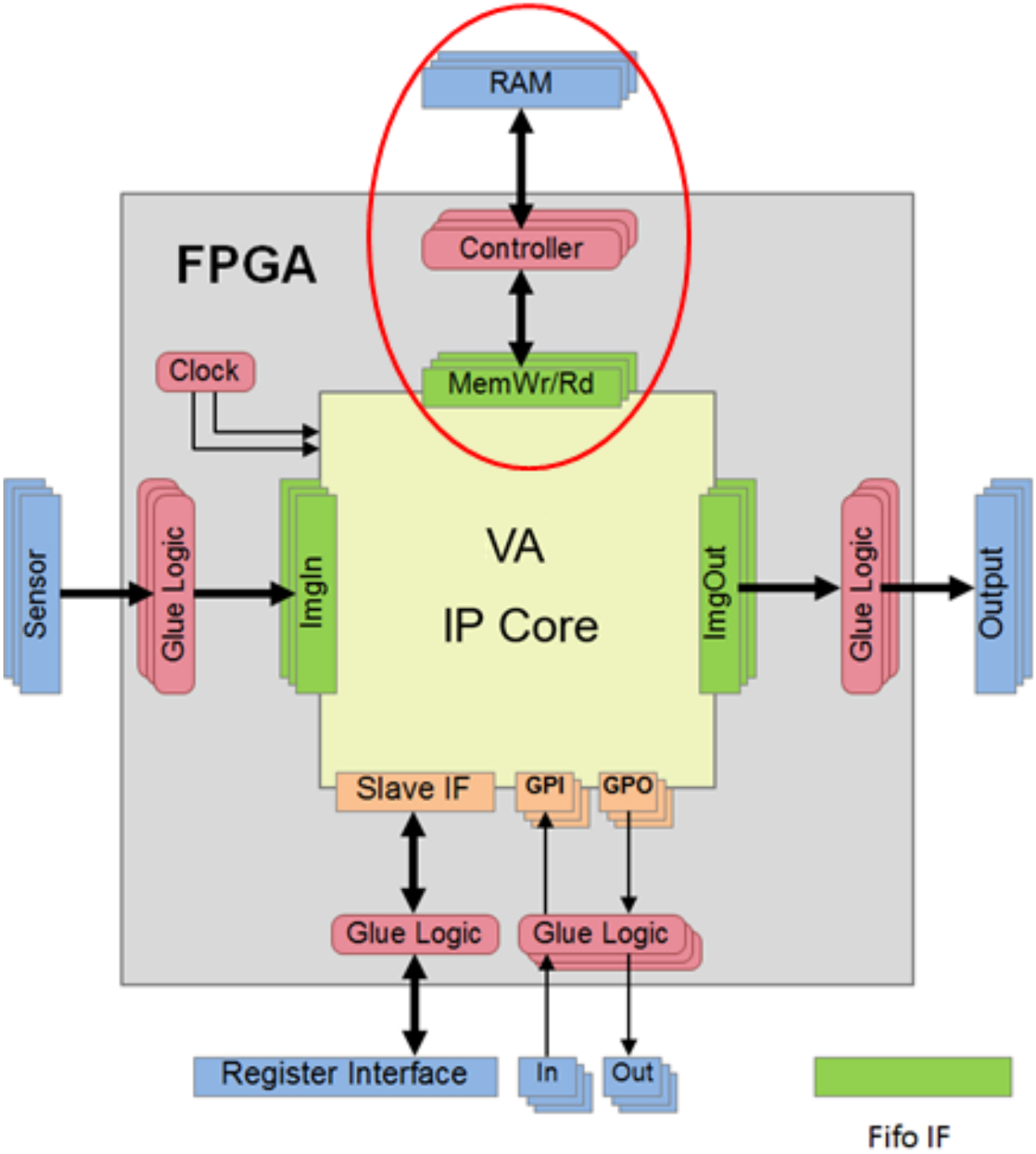
You can connect the VA IP Core to external memory via an abstracted memory interface. You can connect any kind of memory via a single interface mechanism. You simply need to adapt the eVA memory interface protocol via glue logic to the memory controller you use.

You can configure how many memory interface ports will be available on the VA IP Core. For each port, you can define the interface properties individually (i.e., address and data width).



### 5.3.8.1. The Memory Interface of the eVA IP Core

When image buffer operators shall be used VisualApplets need access to external memory. In most cases this will be a number of RAM banks. The memory interface ports are not visible within the VisualApplets design entry. But operators which use external memory (like *ImageBuffer*) consume an abstract resource called **RAM** handled by the resource management of VisualApplets.




The eVA memory interface has been designed in a way that allows to connect a variety of different memory architectures via a single interface mechanism. The abstracted interface is basically a slave interface where the external memory controller masters the access. The slave is demanding write or read accesses via independent FIFO interfaces for writing and reading commands. The attached master needs to acknowledge fetching a command and can perform the access at any time. A flag interface enables the slave to notice when a demanded access has been performed. There are optional signals

for additional information, like the number of commands which are waiting to be fetched. Those signals may help the memory controller master to optimize memory access.

The eVA framework is able to operate with multiple memory ports. The I/O ports of the resulting VA IP Core get a suffix **X** where **X** is the number of the memory interface channel.

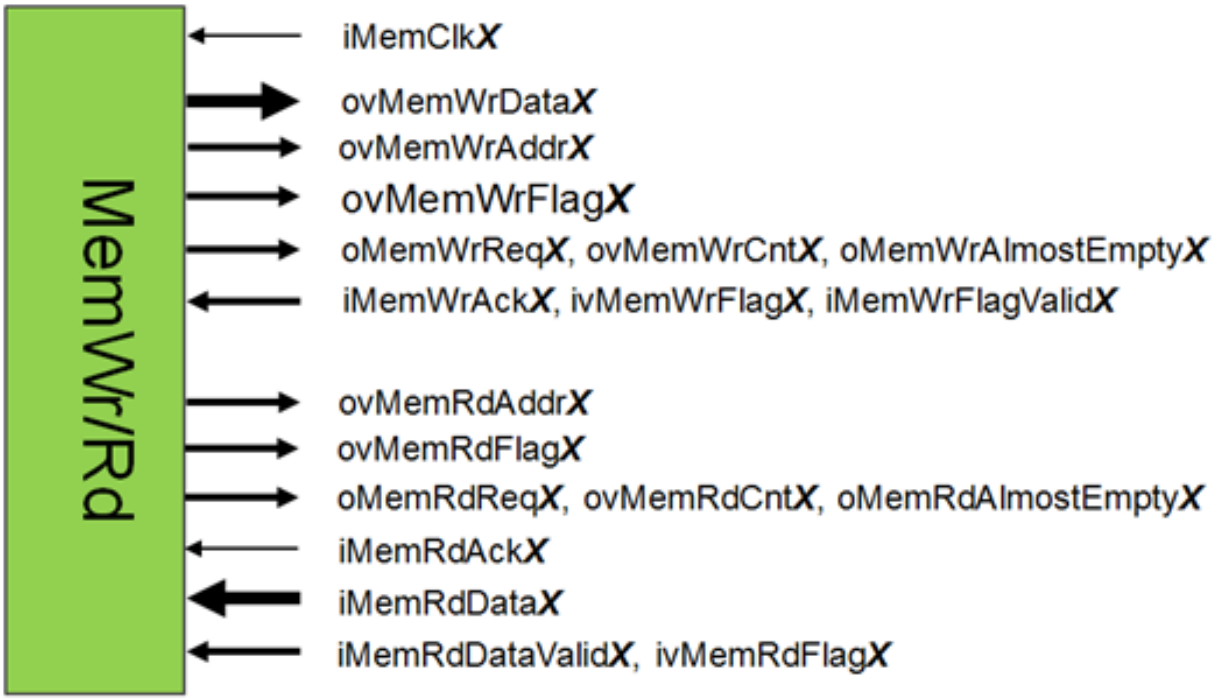



Figure 5.15. Port Layout for Memory Interface Where X Is the Index of the Interface Port

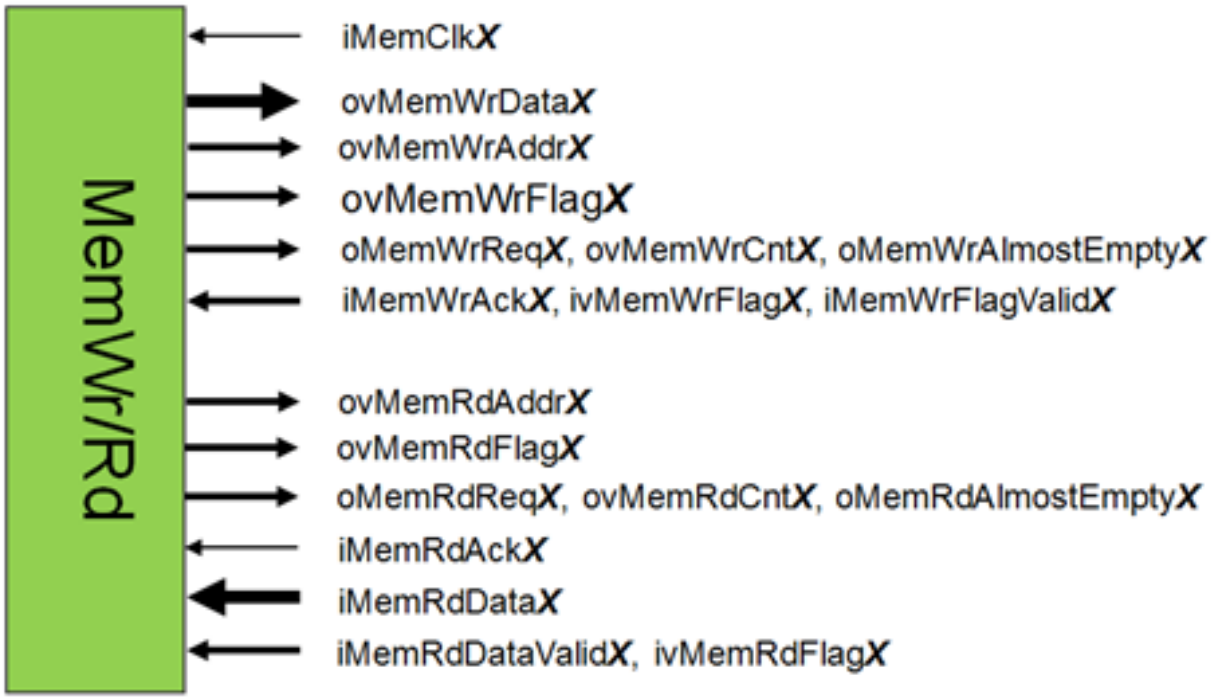
- All I/O is synchronous to the externally supplied clock iMemClk**X** when an asynchronous interface is specified. Alternatively, the synchronization mode can be *SyncToDesignClk* where all I/O is synchronous to *iDesignClk*, or *SyncToDesignClk2x* with I/O synchronous to *iDesignClk2x*.



***iDesignClk2x* Recommended**

Basler recommends to use the *iDesignClk2x* clock signal on memory ports.

- Write and read requests can occur concurrently; write and read data paths to memory are completely separated.
- Write accesses are demanded using the ports *ovMemWrDataX*, *ovMemWrAddrX*, *ovMemWrFlagX*, and *oMemWrReqX* which in the following shall be called a write command.
- Read accesses are demanded using the ports *ovMemRdAddrX*, *ovMemRdFlagX*, *oMemRdReqX* which in the following shall be called a read command.
- The controller acknowledges when an access command has been fetched. After an acknowledge (*iMemWrAckX* for write, *iMemRdAckX* for read), in the next clock tick the next command may be provided. The command is signalled by the according request ports (*oMemWrReqX* for write, *oMemRdReqX* for read).
- Some VA operators require that accesses are tagged by a flag (*ovMemWrFlagX*, *ovMemRdFlagX*). After the access has been performed, the controller must return this flag to the memory interface (*ivMemWrFlagX*, *ivMemRdFlagX*). There is a minimal width of the tags which must be provided:
  - `MemWrFlagWidth >= 4`
  - `MemRdFlagWidth >= 8`




The following table provides a detailed description of the memory interface ports where **X** is a port number for differentiating several memory ports:

| Port | Direction | Width | Description |
|---|---|---|---|
| iMemClkX | In | 1 | Memory interface clock. This input is ignored when the port is configured for synchronous communication. |
| ovMemWrData**X** | Out | MemDataWidth**X** | Write data output to memory controller |
| ovMemWrAddr**X** | Out | MemAddrWidth**X** | Write address |
| ovMemWrFlag**X** | Out | MemWrFlagWidth**X** | Write flag output |
| oMemWrReq**X** | Out | 1 | Write Request |
| iMemWrAck**X** | In | 1 | Acknowledge that write data is taken by the memory controller |
| oMemWrAlmostEmpty**X** | Out | 1 | Only single write command available |
| ovMemWrCnt**X** | Out | MemWrCntWidth**X** | Number of available write commands |
| ivMemWrFlag**X** | In | MemWrFlagWidth**X** | Write flag output from the controller |
| iMemWrFlagValid**X** | In | 1 | Write flag input valid – signals that iMemWrFlag**X** is valid, which means that write access which had been marked with corresponding oMemWrFlag**X** has been executed. |

| Port | Direction | Width | Description |
|---|---|---|---|
| ovMemRdAddr**X** | Out | MemAddrWidth**X** | Read address output |
| ovMemRdFlag**X** | Out | MemRdFlagWidth**X** | Read flag output |
| oMemRdReq**X** | Out | 1 | Read Request |
| iMemRdAck**X** | In | 1 | Acknowledge that read address has been taken by the memory controller |
| oMemRdAlmostEmpty**X** | Out | 1 | Only single read address available |
| ovMemRdCnt**X** | Out | MemRdCntWidth**X** | Number of available read addresses |
| ivMemRdFlag**X** | In | MemRdFlagWidth**X** | Read flag input – only valid when iMemRdDataValid**X** is asserted |
| ivMemRdData**X** | In | MemDataWidth**X** | Read data input |
| iMemRdDataValid**X** | In | 1 | Read data valid |

The following figure illustrates the interface protocol for a write and read access where both accesses overlap. For simplicity, a memory controller with a fixed read latency of two clock cycles has been assumed:

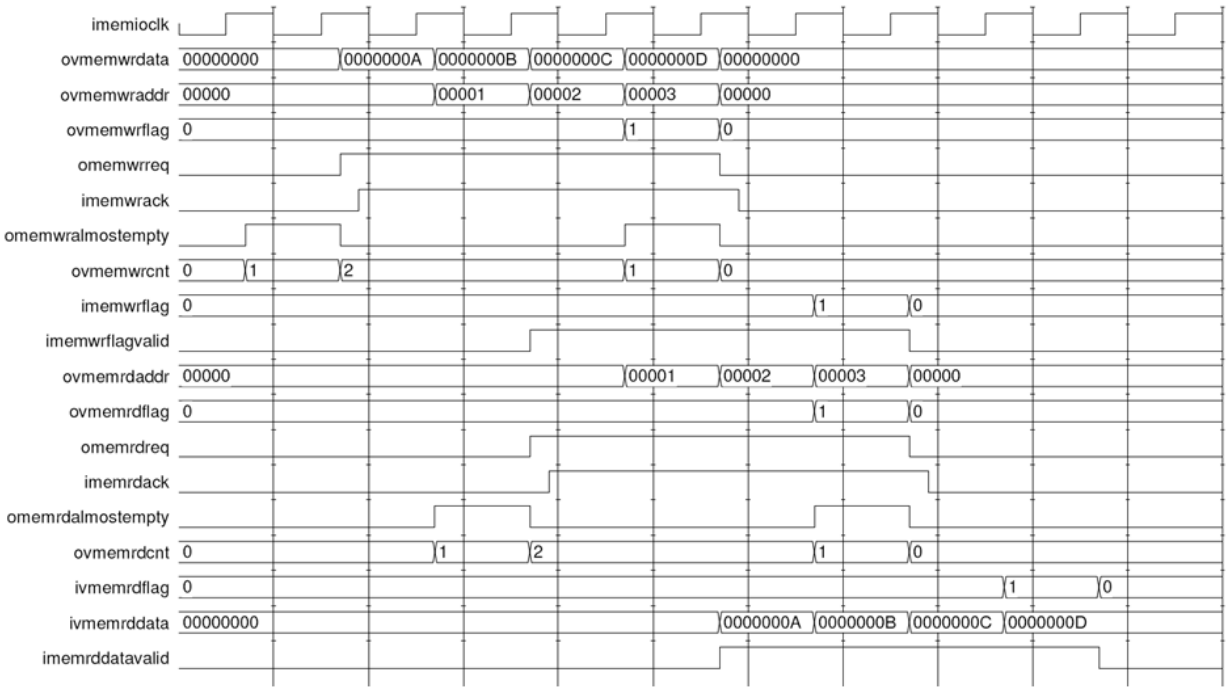



Figure 5.16. Waveform Illustrating the Memory Interface Protocol

## 5.3.8.2. Defining Memory Interfaces

You can define various classes of memory interfaces. Those interface classes may differ in supported data width, address, used clock signals, and in many other aspects. Each memory interface class can be available more than one time on the VA IP Core.

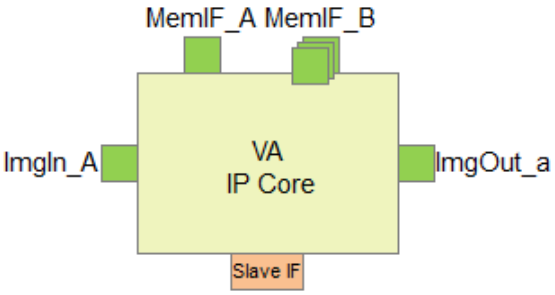
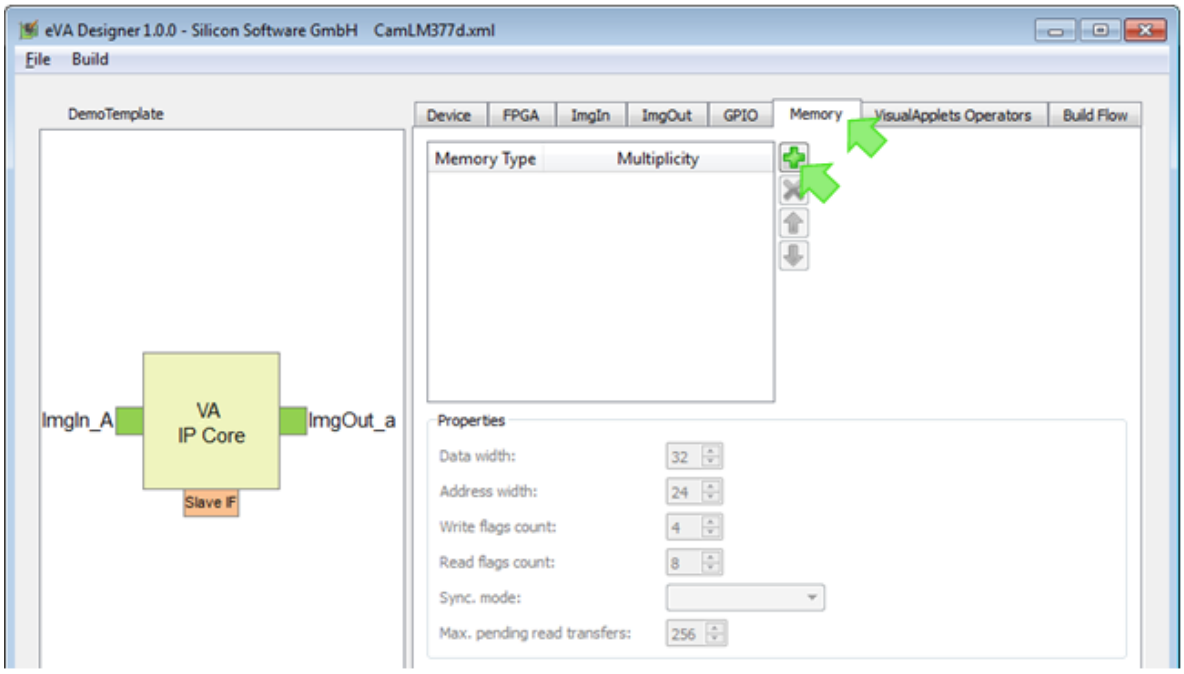



Figure 5.17. Example of VA IP Core

The example VA IP Core (in the figure above) shows two memory interface classes. Multiple instances of both classes are available: This VA IP Core provides 1 memory interface of interface class MemIF_A and 3 memory interfaces of interface class MemIF_B.

To define the memory interfaces of your VA IP Core:

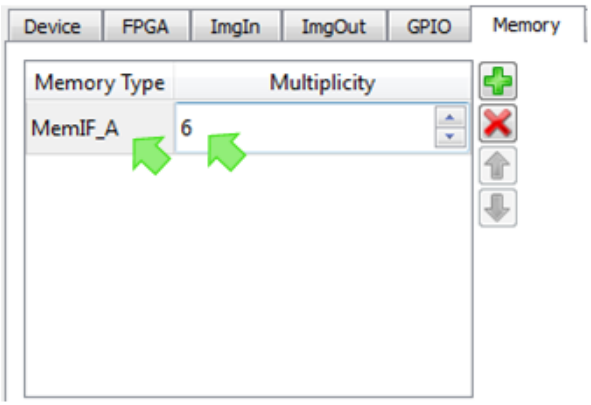
1. Go to the **Memory** tab.



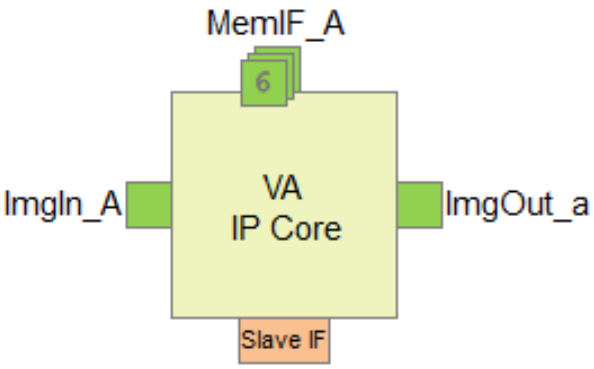

2. Click the plus  icon.

3. Into field **Memory Type**, enter the name of the memory interface class you want to define. Double-click into the field to write.

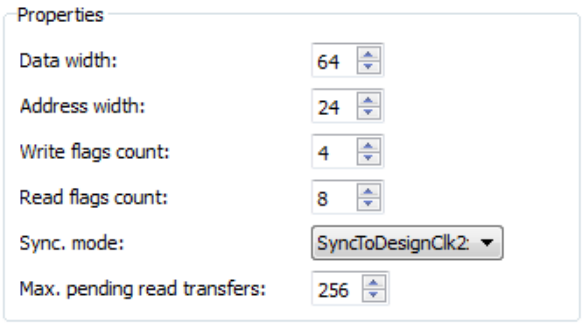
4. In field **Multiplicity**, define how many memory interfaces of this class you want to have on your VA IP Core. Double-click into the field to write.




The defined memory interfaces (in our example, 6 interfaces of class MemIF_A) are immediately visible in the graphical representation of the IP Core:




5. Define further properties of the memory interface class:




**Data width:** Enter the data width for the memory interface class.

**Address width:** Enter here the address width.

**Write flags count:** Enter here the width of ovMemWrFlagX (minimum width: 4)

**Read flags count:** Enter here the width of *ovMemRdFlagX* (minimum width: 8)

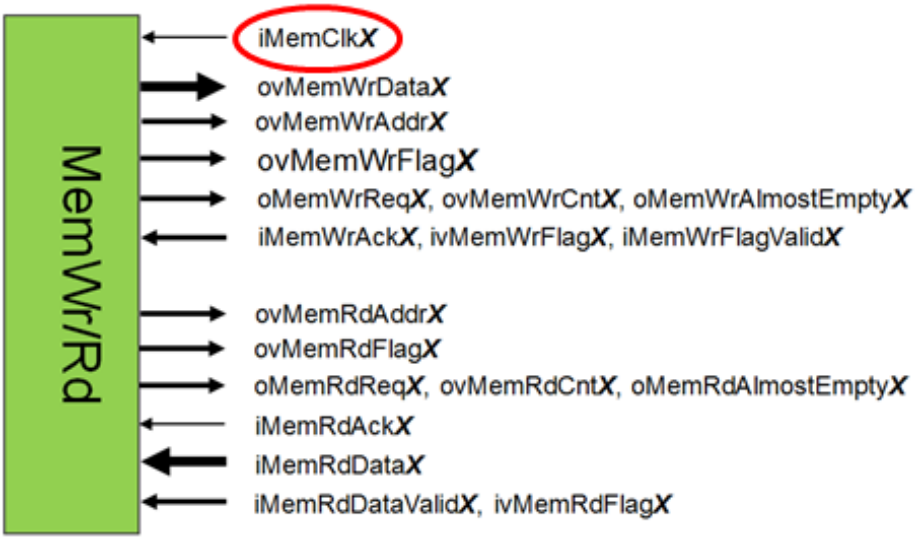
Some VisualApplets operators require that accesses are tagged by a flag (ovMemWrFlagX, ovMemRdFlagX). After the access has been performed, the controller must return this flag to the memory interface (ivMemWrFlagX, ivMemRdFlagX). There is a minimal width of the tags which must be provided: MemWrFlagWidth >= 4, MemRdFlagWidth >= 8

**Sync. mode:** This parameter signals the relation of the memory interface clock (*iMemClkX* on the memory interface) and the design clock (*iDesignClk* on the IP Core). Following values are possible:

- *SyncToDesignClk* – interface ports work synchronous to *iDesignClk*. The external clock input to the interface clock port is ignored.
- *SyncToDesignClk2x* – interface ports are synchronous to *iDesignClk2x*. The external clock input to the interface clock port is ignored.

> **_iDesignClk2x_ Recommended**
>
> Basler recommends to use the *iDesignClk2x* clock signal on memory ports.

- *Async* – asynchronous interface: The interface ports are synchronized to the external clock input to clock port *iMemClkX*.




If you select *Async*, you need to

- provide a clock signal at the *iMemClkX* port of the memory interface, and to
- define the clock domain switch in the synthesis constraints (constraints file).

**Max. pending read transfers:** Enter here the maximum number of data words in the read pipeline outside the core. This value is relevant for calculating the minimum buffer depth for incoming read data.

6. Repeat steps 2 – 5 to define all memory interface classes you need.

## 5.3.9. Defining Hardware-Specific Operators

VisualApplets offers an amount of operator libraries for defining image processing applications. All libraries – except one – are hardware independent, i.e., the contained operators can be used in any design for any target hardware.

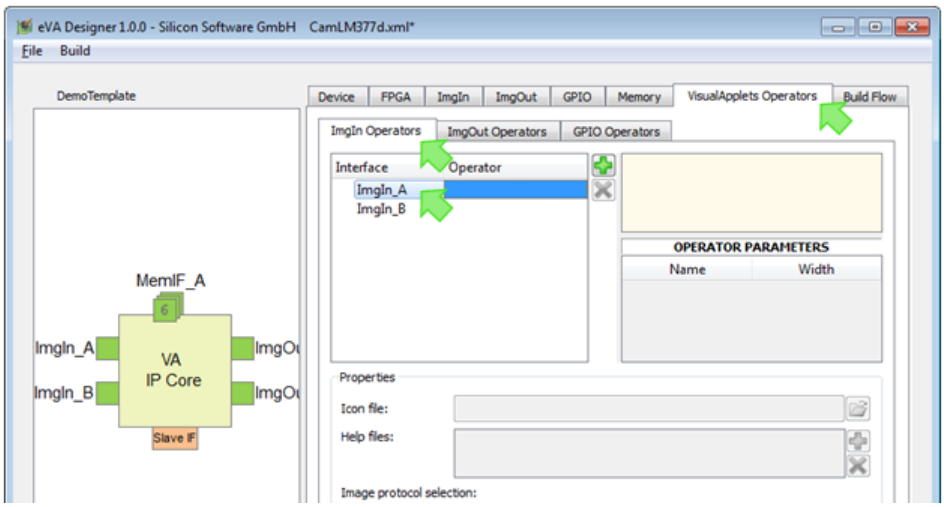
The only exception are operators which connect directly to the hardware that surrounds the VA IP Core: The *ImgIn* interfaces, the *ImgOut* interfaces, and the memory interfaces. These are hardware-specific operators, and they have to be designed together with the IP Core itself.

### 5.3.9.1. Defining *ImgIn* Operators

To define the *ImgIn* operators for your target hardware:

1. Go to the **VisualApplets Operators** tab.

2. Go to the **ImgIn Operators** tab.



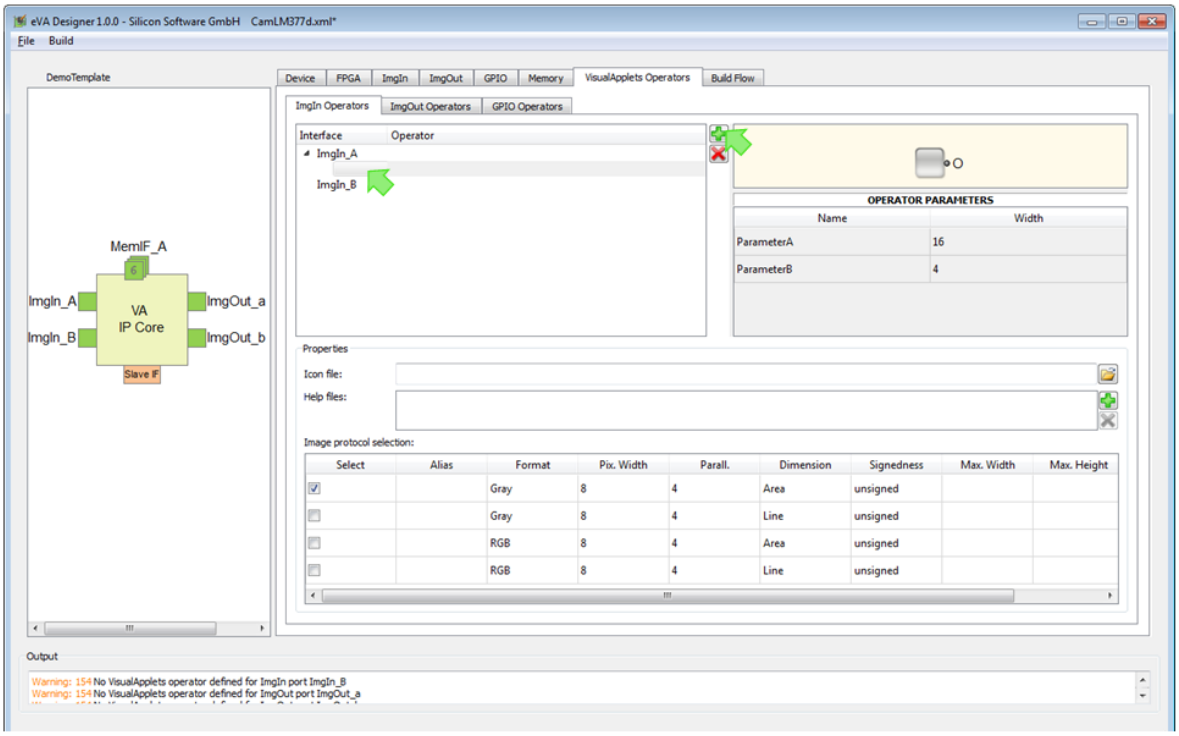

Under tab **ImgIn Operators**, you find a list of the *ImgIn* interface classes you have defined earlier.

(In our example, these are *ImgIn_A* and *ImgIn_B*.)

3. Click on the *ImgIn* interface class for which you want to define connecting *ImgIn* operators.

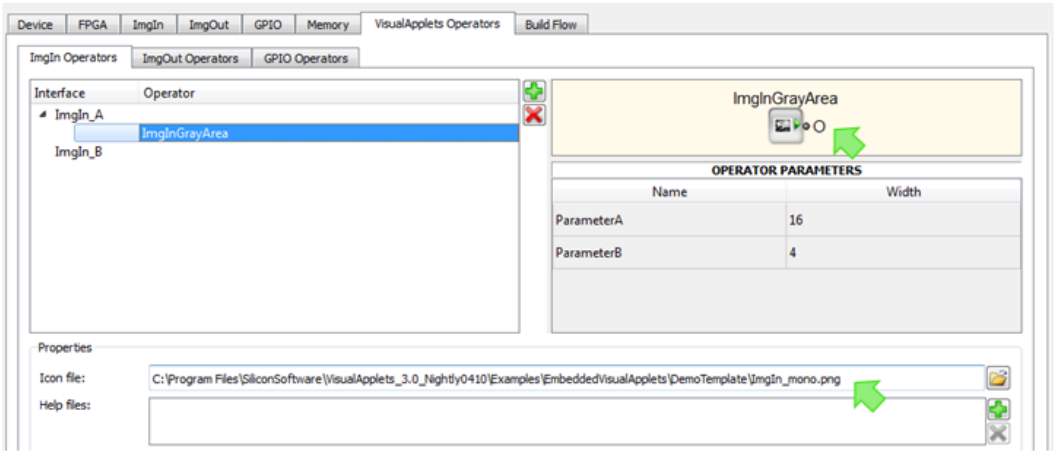
4. Click the plus  icon.

Now, the options for defining operators for this *ImgIn* interface class are displayed:

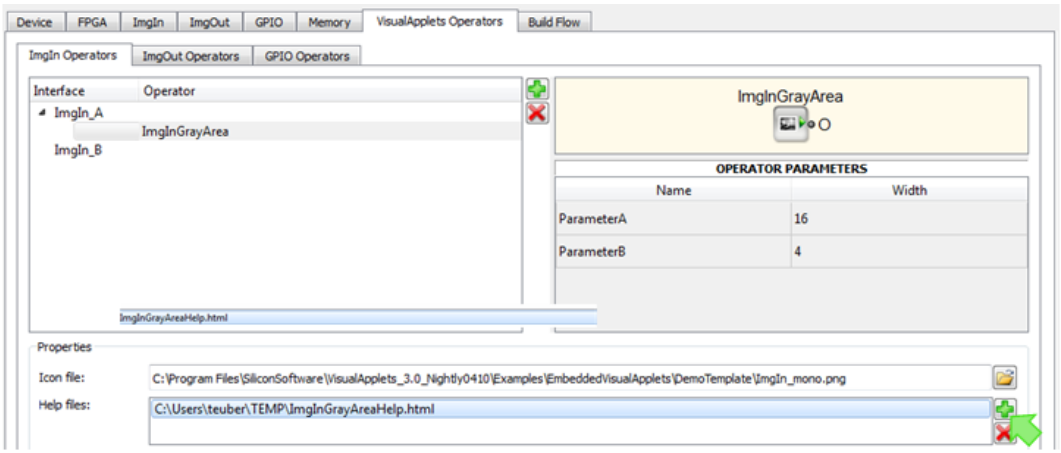



5. Into field **Operator**, enter the name of the operator you want to define. Double-click into the field to write.

6. In field **Icon file**, define path to an icon file that will be displayed in VisualApplets on the graphical representation of the operator. How the complete graphical representation will look like in VisualAplets you can see in the right upper panel of the program window:

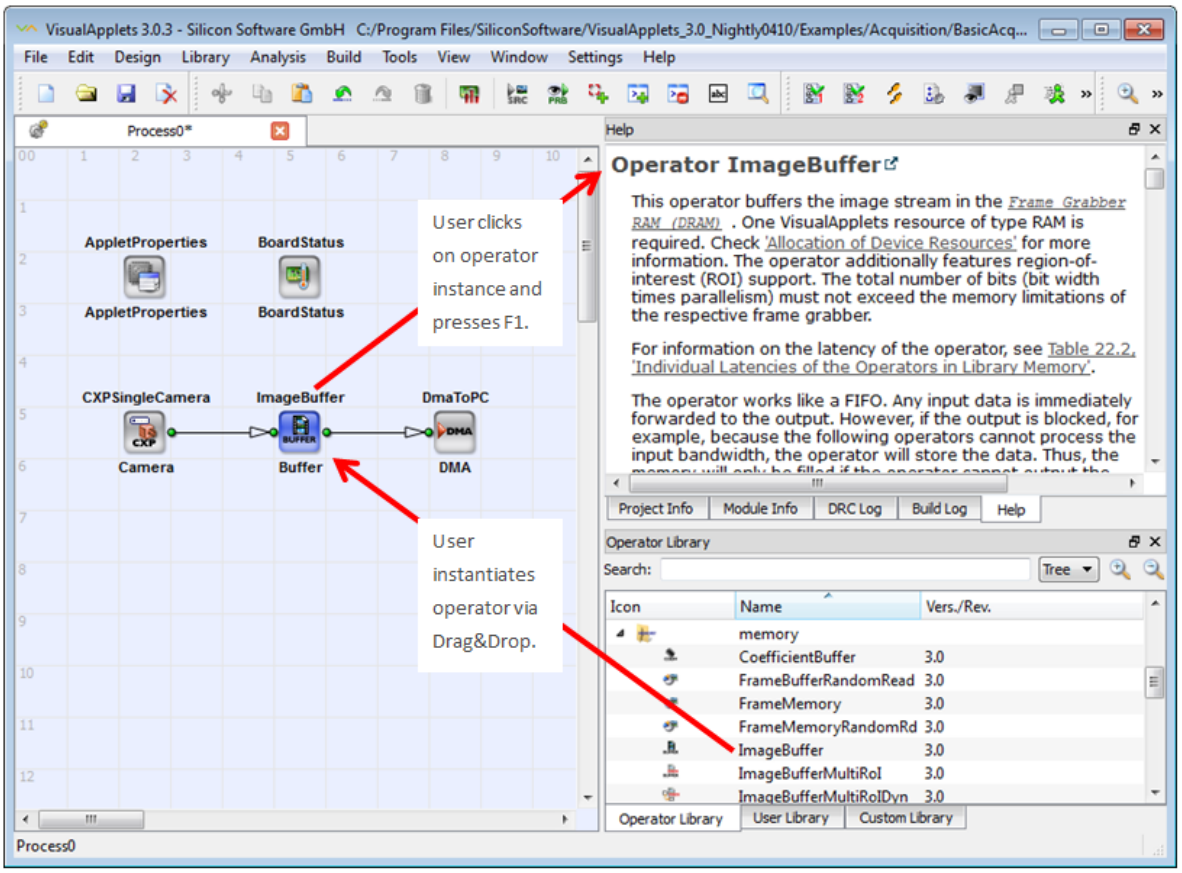



7. In field **Help file**, define path to an HTML help file that describes the operator and its parameters. Click the plus icon to select a file from your file system. The name of the start `*.html` file must be named `<nameofOperator>.html` and be placed in the first place. Other files may be images, CSS files, etc.

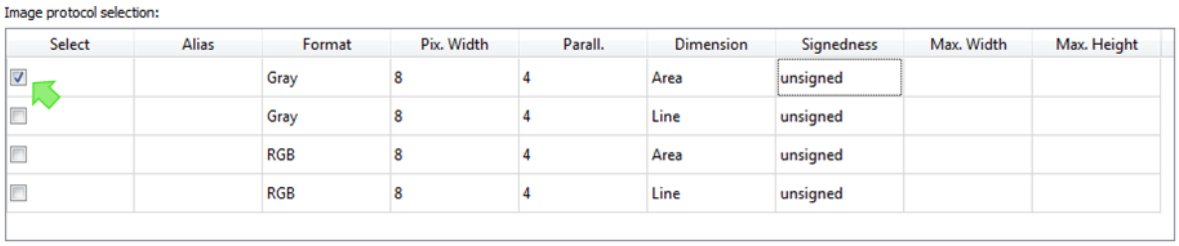



8. Repeat step 7 if you want to add more than one file containing information on this operator. You can load a batch of files (i.e., a HTML file system with graphics files) in one step. The first HTML file you list here will be regarded as the main help file (starting point for help file system).

   The main help file will be visible in the VisualApplets help panel as soon as a user instantiates an operator, clicks on the operator instance in the design window and presses F1:




Panel **Image protocol selection** displays the image protocols attached to the selected interface class:

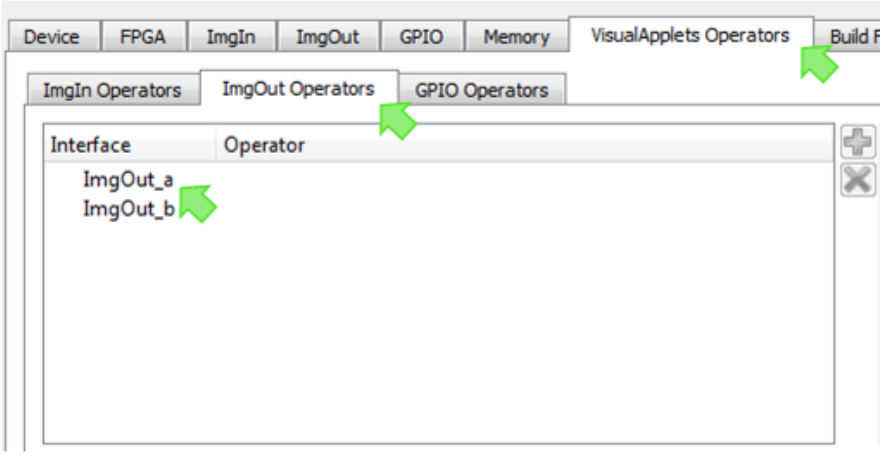



9. Select here the image protocol(s) you want this operator to support.
10. If you want to, define maximal image width and maximal image height for a selected image protocol in the two last columns.
11. Repeat steps 4 – 10 to define additional operators being connectible to the selected interface class.
12. Repeat steps 3 – 10 to define operators for the other interface classes.

### 5.3.9.2. Defining *ImgOut* Operators

To define the *ImgOut* operators for your target hardware:

1. Go to the **VisualApplets Operators** tab.
2. Go to the **ImgOut Operators** tab.

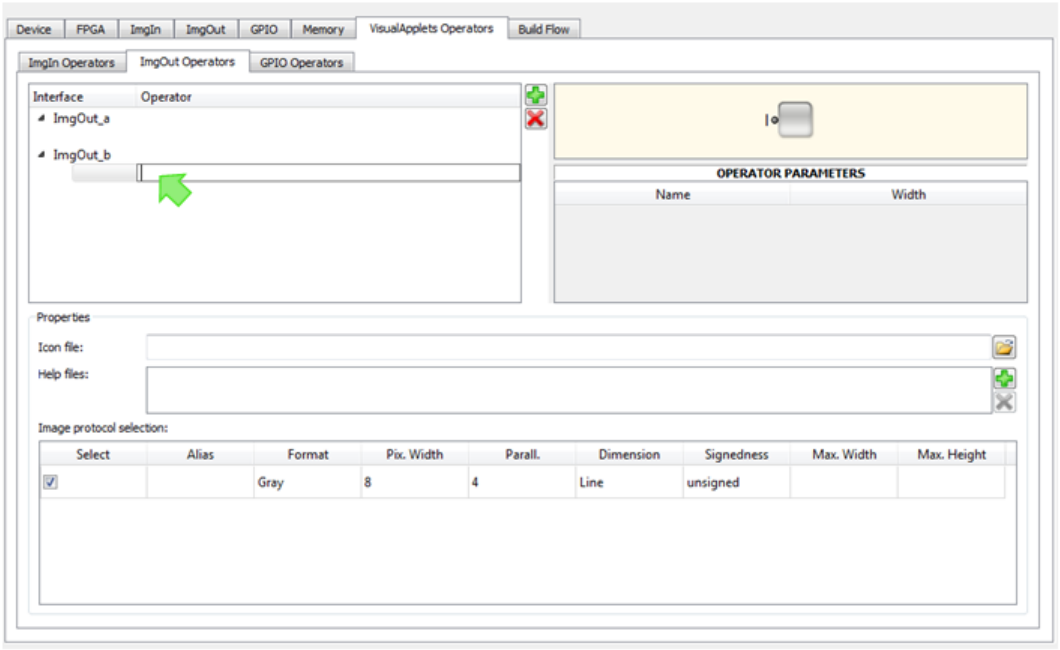



Under tab **ImgOut Operators**, you find a list of the *ImgOut* interface classes you have defined earlier. (In our example, these are *ImgOut_a* and *ImgOut_b*.)

3. Click on the *ImgOut* interface class for which you want to define connecting *ImgOut* operators.

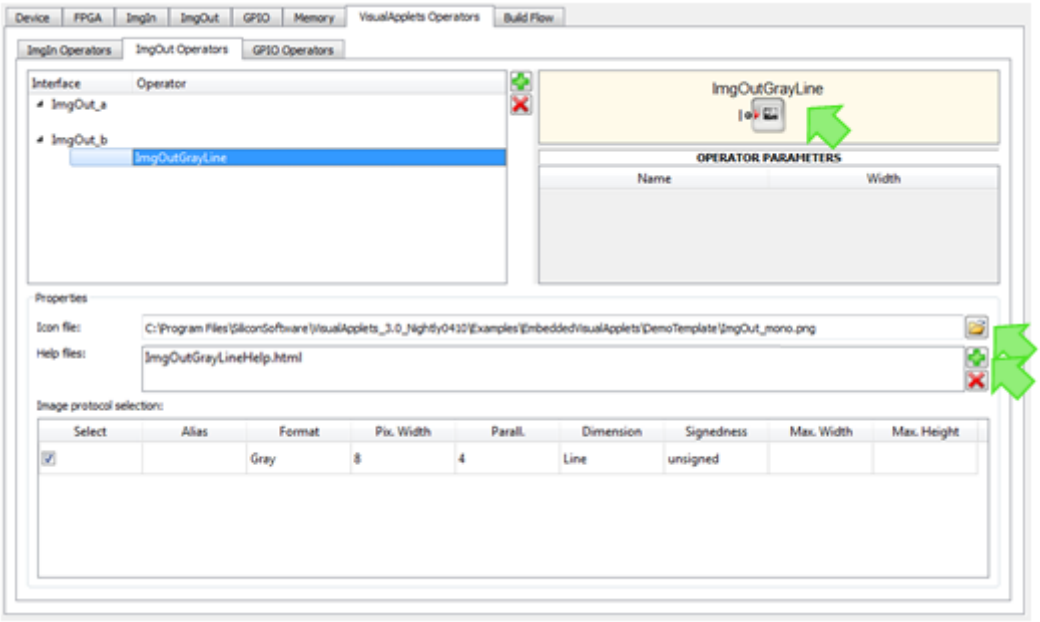
4. Click the plus  icon.

   Now, the options for defining operators for this *ImgOut* interface class are displayed:




5. Into field **Operator**, enter the name of the operator you want to define. Double-click into the field to write.

6. In field **Icon file**, define the path to an icon file that will be displayed in VisualApplets on the graphical representation of the operator. How the complete graphical representation will look like in VisualAplets you can see in the right upper panel of the programm window:




7. In field **Help file**, define path to an HTML help file that describes the operator and its parameters. Click the plus icon to select a file from your file system.

8. Repeat step 7 if you want to add more than one file containing information on this operator. You can load a batch of files (i.e., a HTML file system with graphics files) in one step. The first HTML file you list here will be regarded as the main help file (starting point for help file system).

   The main help file will be visible in the VisualApplets help panel as soon as a user instantiates an operator, clicks on the operator instance in the design window, and presses F1.

   Panel **Image protocol selection** displays the image protocols attached to the selected interface class:

Image protocol selection:

| Select | Alias | Format | Pix. Width | Parall. | Dimension | Signedness | Max. Width | Max. Height |
|---|---|---|---|---|---|---|---|---|
| ☑ | | Gray | 8 | 4 | Area | unsigned | | |
| ☐ | | Gray | 8 | 4 | Line | unsigned | | |
| ☐ | | RGB | 8 | 4 | Area | unsigned | | |
| ☐ | | RGB | 8 | 4 | Line | unsigned | | |

9. Select here the image protocol(s) you want this operator to support.

10. If you want to, define maximal image width and maximal image height for a selected image protocol in the two last columns.

11. Repeat steps 4 – 10 to define additional operators being connectible to the selected interface class.

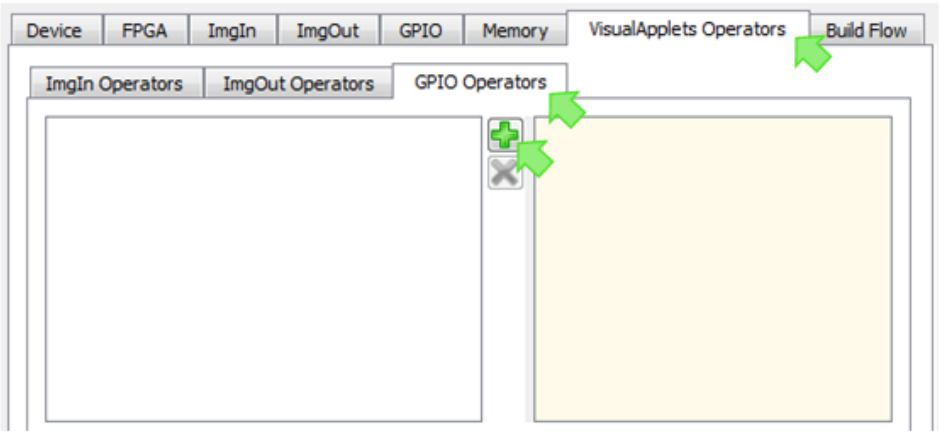
12. Repeat steps 3 – 10 to define operators for the other interface classes.

### 5.3.9.3. Defining GPIO Operators

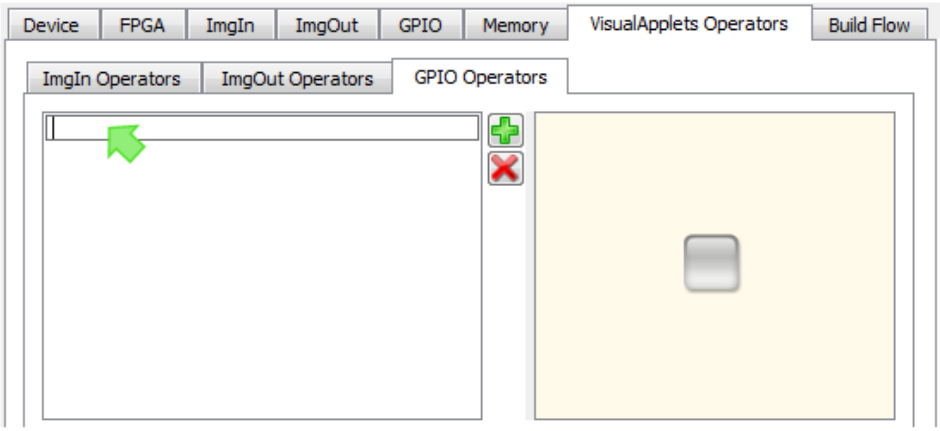
You can define GPIO operators that provide an configurable number of GPI and GPO ports. When designing image processing applications, the user can connect these ports to the actual GPI and GPO ports of the eVA IP Core.

To define GPIO operators for your target hardware:

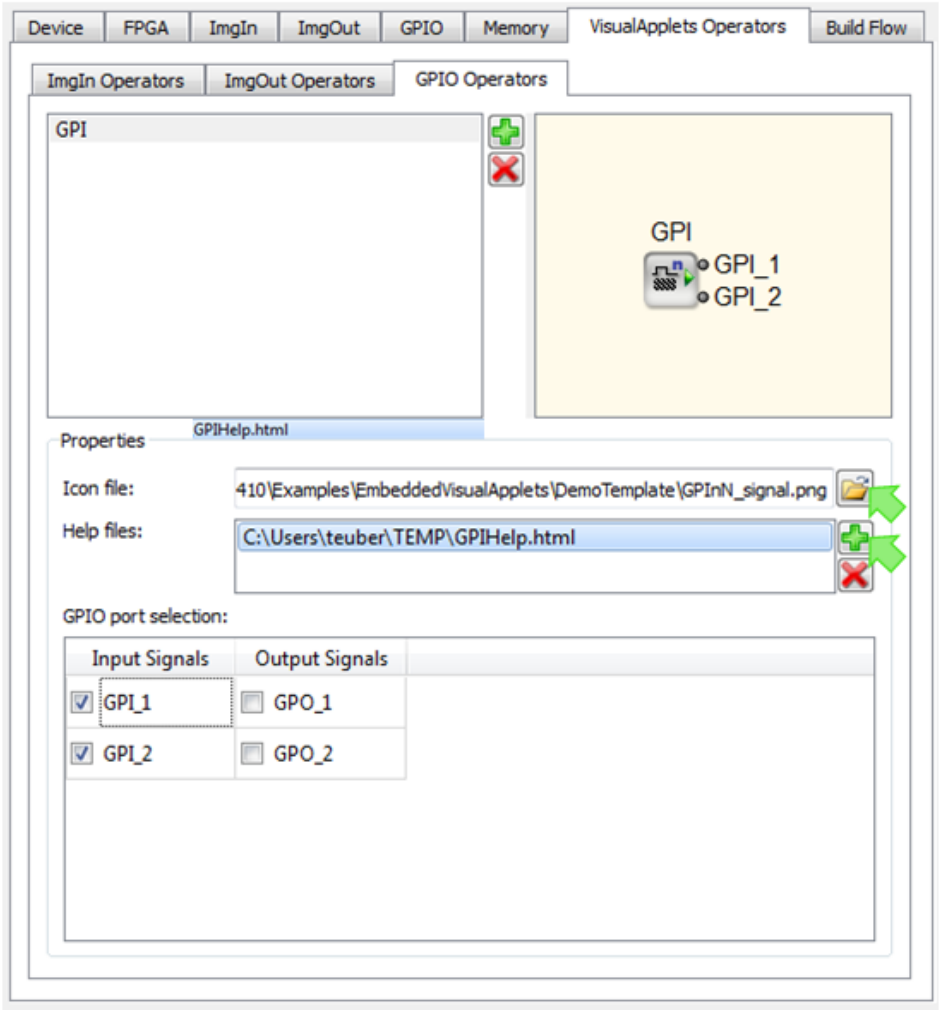
1. Go to the **VisualApplets Operators** tab.
2. Go to the **GPIO Operators** tab.
3. Click the plus  icon.




4. Into the field that opens, enter the name of the operator you want to define. Double-click into the field to write.



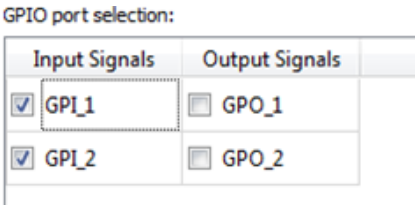

5. In field **Icon file**, define path to an icon file that will be displayed in VisualApplets on the graphical representation of the operator. How the complete graphical representation will look like in VisualAplets you can see in the right upper panel of the programm window.
6. In field **Help file**, define path to an HTML help file that describes the operator and its parameters. Click the plus icon to select a file from your file system.




7. Repeat step 6 if you want to add more than one file containing information on this operator. You can load a batch of files (i.e., a HTML file system with graphics files) in one step. The first HTML file you list here will be regarded as the main help file (starting point for help file system).

   The main help file will be visible in the VisualApplets help panel as soon as a user instantiates an operator, clicks on the operator instance in the design window, and presses F1.

   Panel **GPIO port selection** displays the GPI and GPO interfaces you defined for the eVA IP Core:




8. Select here the GPI and GPO ports of the IP core you want this operator to be able to connect to.

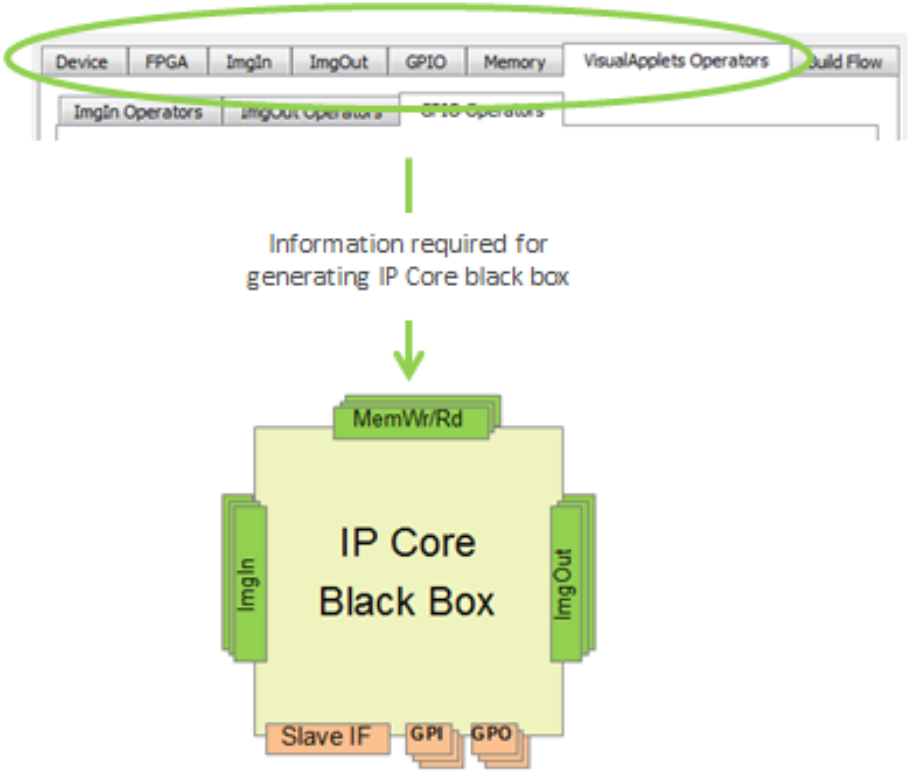
9. Repeat steps 3 – 8 to define additional operators being connectible to GPI and GPO ports of the IP core.

## 5.3.10. Generating VHDL Code for the IP Core

After you defined the IP Core black box as described in sections Section 5.2, 'Common Interfaces for all Platforms' to Section 5.3.9, 'Defining Hardware-Specific Operators', you can build the IP Core black box (VHDL). In Section 5.4, 'Embedding and Simulating the IP core', you will integrate this black box into you your FPGA design.

Together with the IP Core black box, **eVA Designer** automatically generates a simulation environment for the individual interfaces you defined, and an according test bench. Section Section 5.4, 'Embedding and Simulating the IP core' describes in detail how you can use them.

For generating the IP core black box, you do not need to specify any build flow details. Also the hardware-specific operators you may define at a later stage.

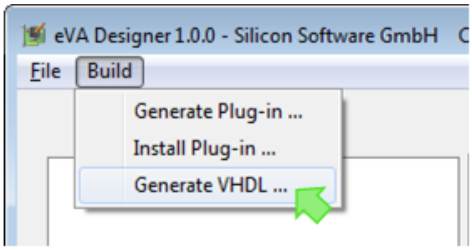



To generate the IP Core black box:

1. Enter all information regarding your target hardware and the required IP Core interfaces as described in sections Section 5.2, 'Common Interfaces for all Platforms' to Section 5.3.9, 'Defining Hardware-Specific Operators'.

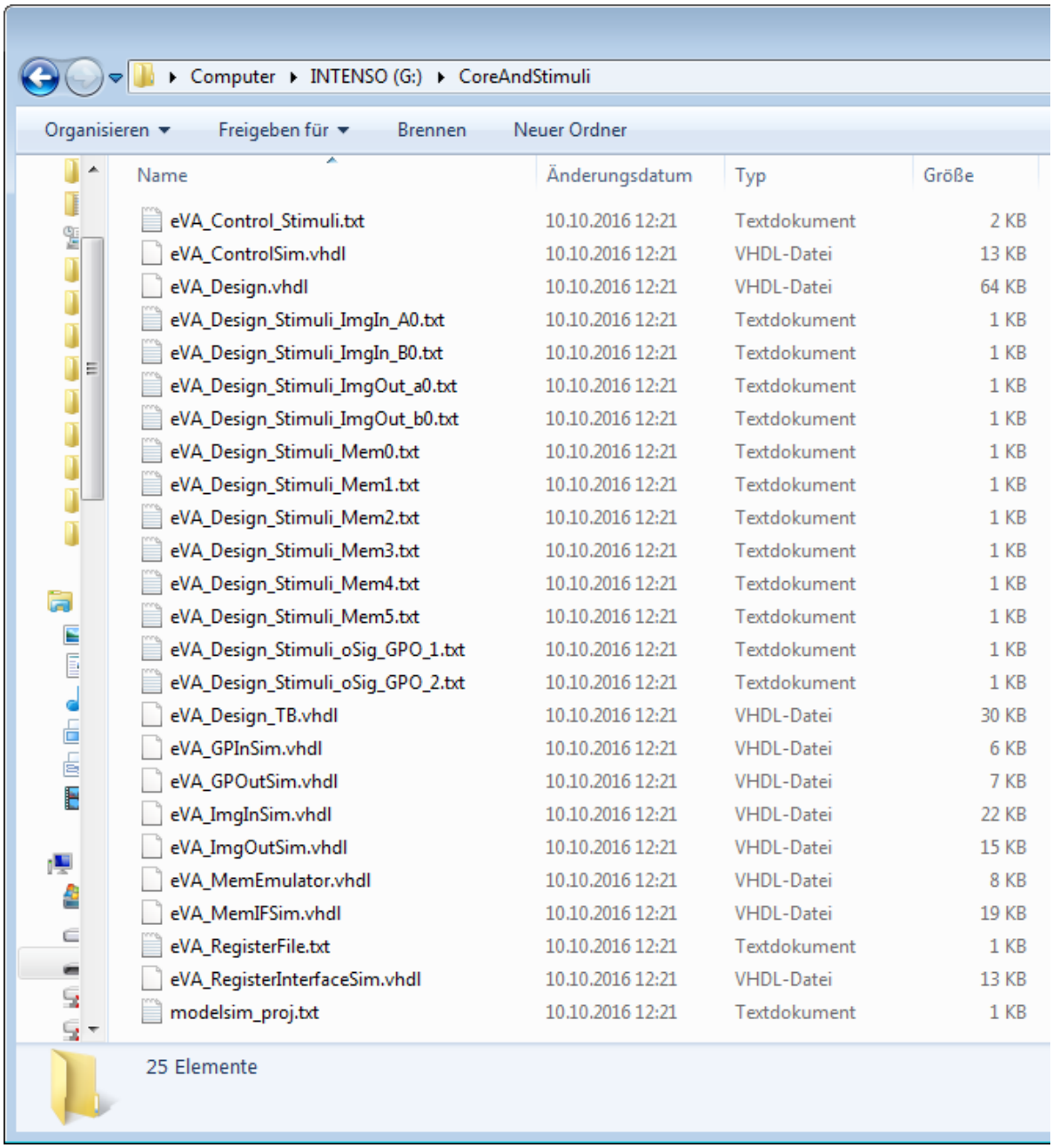
2. Open the XML or `*.eva` file containing your hardware and core interface description.

3. From menu **Tools**, select **Generate VHDL ...**




4. Create a dedicated directory where you will to store the generated files.

5. Click on **Select Directory**. Now, all related files are generated automatically. You find them in the selected directory:



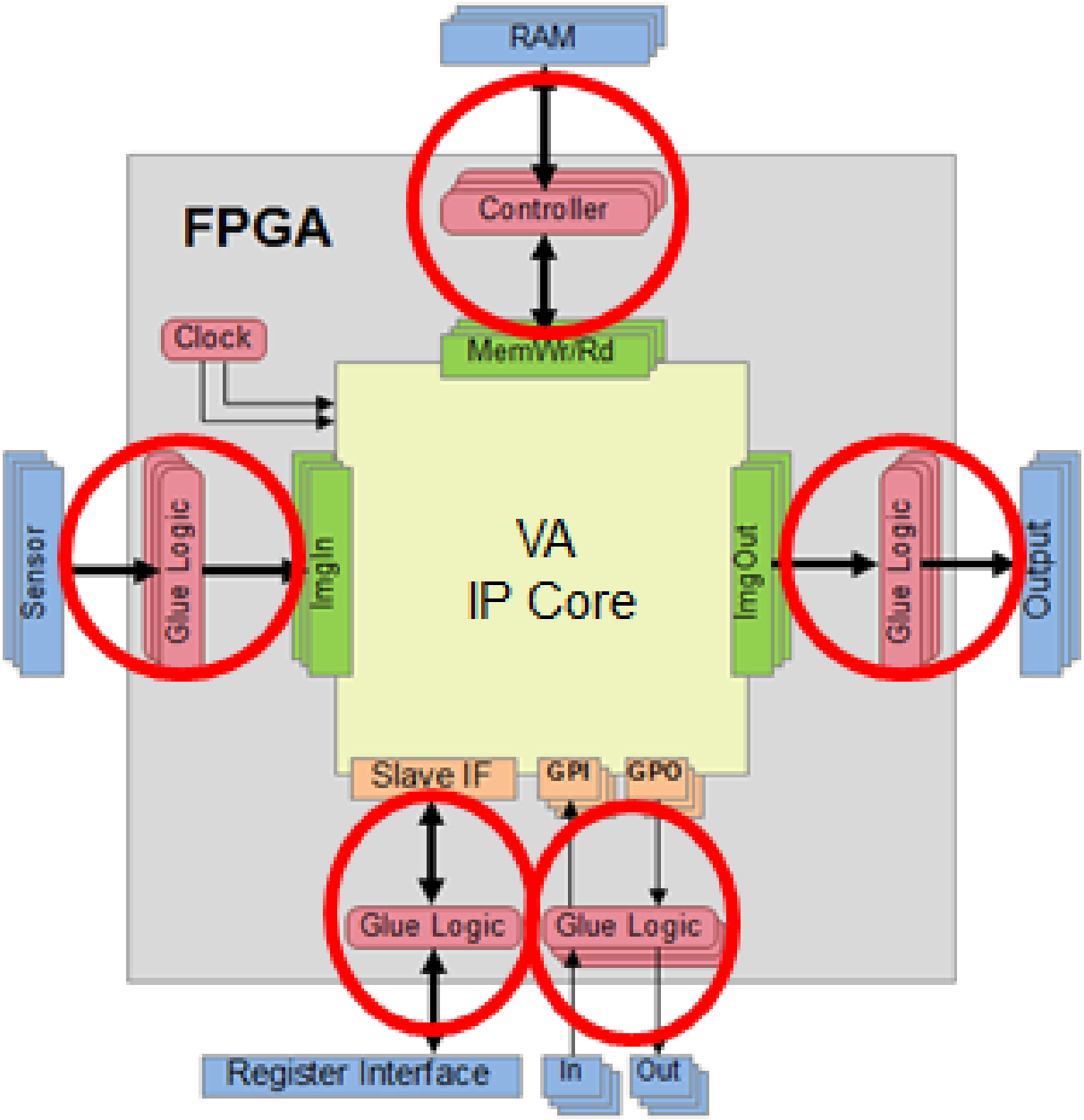

6. Follow the instructions in section Section 5.4, 'Embedding and Simulating the IP core'. You will learn about

   a. integrating the IP core black box (that you've just created) into your FPGA design (VHDL),

   b. using the stimuli and simulation files that support you during VHDL integration.

## 5.4. Embedding and Simulating the IP core

You are ready now to integrate the IP core into your surrounding FPGA design. This basically means implementing and connecting converters between the data communication protocols of the target hardware and the protocols at the IP core interfaces. In particular, you need to care for connecting external interfaces like memory controller, sensor interface and data output.




When you generated the IP core black box (as described in section Section 5.3.10, 'Generating VHDL Code for the IP Core'), not only the IP core black box itself was generated. In the same step, a whole test bench was generated that is already tailored to test the individual interfaces of the IP core while you integrate them into your HDL design.

The test bench enables simulation of data communication through the IP core interfaces during HDL integration. The tools generated for HDL simulation focus on testing the interface protocols and generating stimuli for the circuits attached to the core in the top-level architecture (your HDL design).

In the test bench, each **individual interface port** which may later be connected via a corresponding VisualApplets operator is **simulated independently**, driven by file I/O. The simulation entity consists of the following elements:

- **Emulation of slave interface for register access:** Configured by file, the simulation module provides a set of registers which can be written and read. There are dedicated registers which connect to the enable and reset signals. You can configure different processes. The actual number of processes of an image processing application is later (after integration) defined in the VisualApplets design. For simulation, you define the number of processes as described in section Emulation of Slave Interface. . Enable and Reset are the only signals which connect to the other simulation modules whose affiliation

to a process you can specify through simulation entity parameters (see below). The register interface emulator provides the option to change register values over time according to a stimuli file. In the stimuli file, you can set a value and define when a value should be changed (i.e., set 1, after 10.000 clock cycles set 0).

- Emulator of image source connected to ImgOut ports. Stimulated by file, this kind of module outputs image data to the ImgOut interface of the IP core. The image protocol ID you can configure according to the image protocols you specified for an image output port.
- Emulator of frame sink connected to ImgIn ports. This kind of module emulates an operator which is connected to the ImgIn interface of the IP core. The image protocol ID you can configure according to the image protocols you specified for an image input port. The module writes the received data to file.
- Memory port emulator which acts as if a module is connected which uses RAM. The sequence of write and read accesses is stimulated by a file and read data is output to file.
- GPIO emulator. Each GPIO signal for output is driven by a signal generator which is configured by a file. Each GPIO signal input is monitored and changes of the signal are written to a report file.

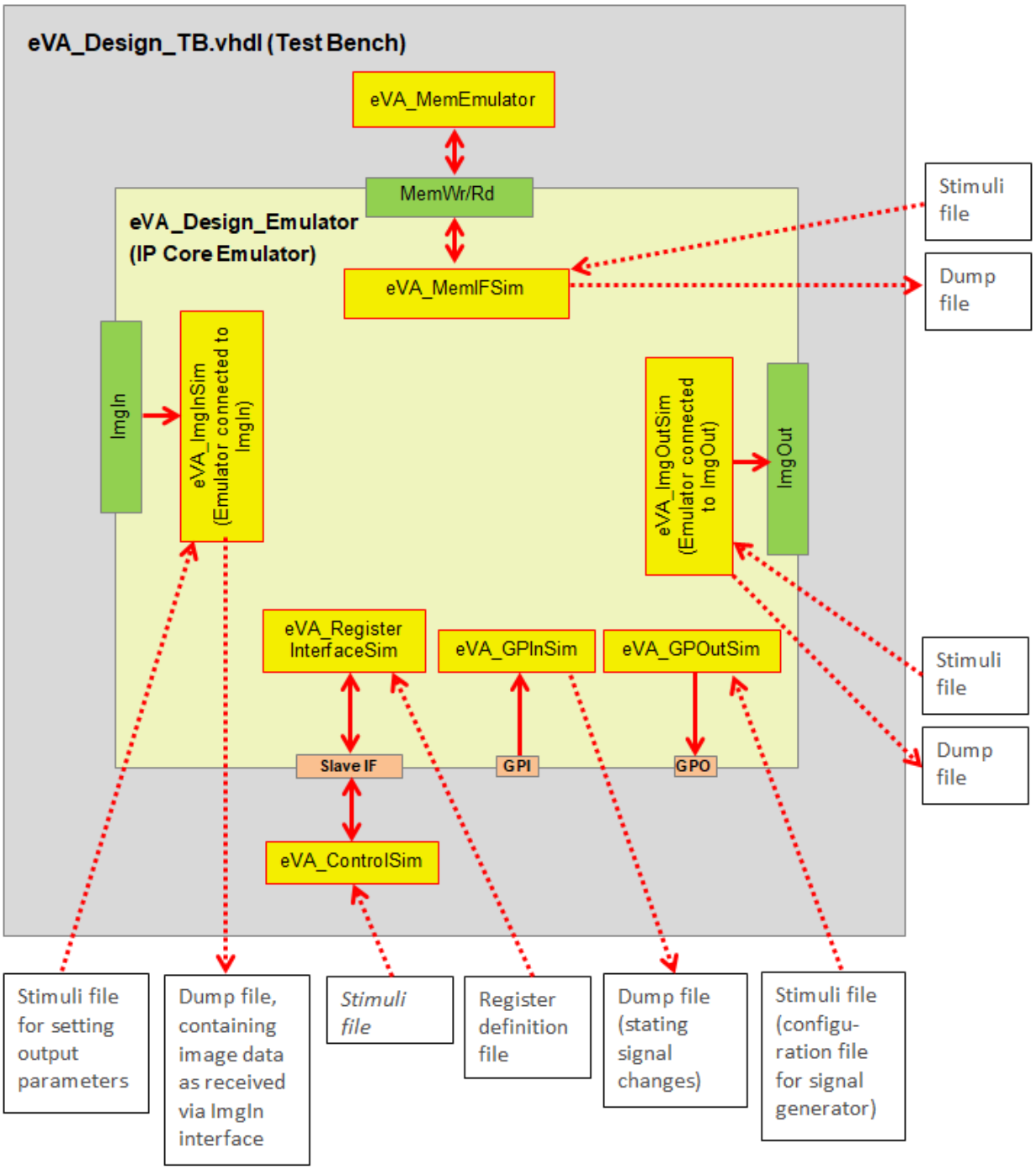



Figure 5.18. Example Test Bench for IP Core with 1 ImgIn Interface, 1 ImgOut Interface, 1 Memory Interface, 1 GPI, 1 GPO, and Slave Interface

### 5.4.1. Simulation Framework

For RTL level simulation, a VHDL file **eVA_Design.vhdl** has been created. This file contains a package with name eVA_<**PLATFORMNAME**> ( <**PlatformName**> is the platform name you defined as described in Section 5.3.3, 'Entering Platform Details'). This package contains two components:

- **eVA_Design**: black box for the Visual Applets IP Core
- **eVA_Design_Emulator**: emulator of Visual Applets IP Core: The emulator implements the file-based stimuli generation for a simulation test bench.

The following shows the code in file **eVA_Design.vhdl** which would be generated for a most simple VA IP Core consisting only of the slave interface for register access:

```
component eVA_Design
generic(

       RegIFDataWidth : integer := 32;
       RegIFAddrWidth : integer := 16);

port(

       iDesignClk: in std_logic := '0';
       iDesignClk2x: in std_logic := '0';

       ivRegWrData: in std_logic_vector(RegIFDataWidth-1 downto 0) := (others=>'0');
       ivRegWrAddr: in std_logic_vector(RegIFAddrWidth-1 downto 0) := (others=>'0');
       iRegWrValid: in std_logic := '0';

       ivRegRdAddr: in std_logic_vector(RegIFAddrWidth-1 downto 0) := (others=>'0');
       iRegRdValid: in std_logic := '0';
       ovRegRdData: out std_logic_vector(RegIFDataWidth-1 downto 0) := (others=>'0');
       oRegRdDataValid: out std_logic := '0'

);
end component;

component eVA_Design_Emulator
generic(

       RegIFDataWidth : integer := 32;
       RegIFAddrWidth : integer := 16;
       NrOfProcesses: integer := 2;
       RegisterDefinitionFile: string := ""

);
port(

       iDesignClk: in std_logic := '0';
       iDesignClk2x: in std_logic := '0';

       ivRegWrData: in std_logic_vector(RegIFDataWidth-1 downto 0) := (others=>'0');
       ivRegWrAddr: in std_logic_vector(RegIFAddrWidth-1 downto 0) := (others=>'0');
       iRegWrValid: in std_logic := '0';

       ivRegRdAddr: in std_logic_vector(RegIFAddrWidth-1 downto 0) := (others=>'0');
       iRegRdValid: in std_logic := '0';
       ovRegRdData: out std_logic_vector(RegIFDataWidth-1 downto 0) := (others=>'0');
       oRegRdDataValid: out std_logic := '0'

);
end component;
```

More detailed information about emulating the individual kinds of IP core interfaces you find in the following subsections.

### 5.4.1.1. Emulation of Slave Interface

The eVA design emulator has a parameter **NrOfProcesses**. This parameter is an integer within the range 1 to 16. The parameter has the effect that a set of predefined registers for resetting and enabling the design are created. Following addresses will be configured:

- 0x00: Global Reset
- 0x01: Global Enable
- 0x02 * (i+1): Process Reset for process i (i < NrOfProcesses).
- 0x02 * (i+1) + 0x01: Process Enable for process i (i < NrOfProcesses).

So when NrOfProcesses = 2 the first 6 addresses of the slave interface address space will be used for controlling reset and enable of the emulated design. Each image port will be affiliated to one pair of reset and enable signals controlled by the above registers. After integration, the VisualApplets user will affiliate an interface to a process, as he instantiates an operator as component of a process and connects this operator instance to an interface.

The emulator for the slave interface for register access is further configured by a text file which is set by the entity parameter RegisterDefinitionFile as provided in the above VHDL code. The parameter is set automatically during black box and test bench generation as described in section Generating VHDL Code for the IP Core.

The following commands may be present in the register definition file:

| Command | Description |
|---|---|
| **REM** | Rest of line is comment |
| **DEF** | Define register. This command has the following syntax:<br>`DEF <addr> <width> <write_read> <init_value>`<br><br>with the parameters:<br><br><init_value>: hexadecimal initial value<br><br><addr>: hexadecimal value of register address<br><br><width>: bit width of register<br><br><write_read>: 1 for write register, 0 for read register |
| **CON** | Connect write register to read register. This command has the following syntax:<br>`CON <wrRegAddr> <rdRegAddr>`<br><br>with the parameters:<br><br><wrRegAddr>: address of write register (hex)<br><br><rdRegAddr>: address of read register (hex) |
| **WCK** | Wait for a number of clock cycles. The syntax is as follows:<br>`WCK <clock_ticks>`<br><br>with <clock_ticks> giving the number of clock ticks in hexadecimal format. |
| **SET** | Set value of read register<br>`SET <rdRegAddr> <value>`<br><br>with the parameters:<br><br><rdRegAddr>: address of read register (hex)<br><br><value>: hexadecimal register value |

After the last parameter of any command, you can add a comment preceded by '#'.

The following code is an example register definition file which configures two write and two read registers, where one read register is preset after 16 clock cycles (address 0x7) and the other read register (address 0x6) is driven by the write register with address 0x6:

```
REM ************************************************************
REM Command formats: DEF <addr> <width> <write_read> <init_value>
```

```
REM                  CON <wrRegAddr> <rdRegAddr>
REM                  WCK <clock_ticks>
REM                  SET <rdRegAddr> <value>
REM ********************************************************
DEF 0006 10 1 00000000 #define write reg with width 0x10 at address 0x6
CON 0006 0006          #create read reg with addr 0x6 connected to
REM                     write register with address 0x6
DEF 0007 20 1 00000000 #define write reg with width 0x20 at address 0x7
DEF 0007 10 0 00000000 #define read register at address 0x7
WCK 0010               #wait for 16 clock cycles
SET 0007 0000000C      #set read register value
```

### 5.4.1.2. Emulation of ImgOut Interface

The emulation of image communication interfaces of type ImgOut is driven by a stimuli file providing information about the sequence of data output.

For any present ImgOut port, the eVA_Design_Emulator entity has a generic **<PORTIDX>_StimuliFileName** where **<PORTIDX>** is the name of the corresponding image output port class followed by the port number. Each line within the given file must follow the syntax

`<Command> <Data> <EndOfLine> <EndOfFrame> <DataValid>`

where **<Command>** is a three letter command, **<Data>** provides an hexadecimal data word, and the three remaining parameters correspond to the image protocol flags.

The following table describes the available commands:

| Command | Description |
|---|---|
| DAT | Data command. This command provides data which will become output at the port ov`PORTIDX`Data and the associated image protocol flag ports. |
| WCK | Wait command. The parameter <Data> provides the number of clock ticks for which the command interpreter pauses. |
| FID | Set FID output. The parameter <Data> provides the value to which the port ov**`PORTIDX`**_FID_D will be set. |
| PDX | Set the X'th port parameter output to the value provided with <Data>, where X may be a number between 0 and 9. If for example the first parameter output of the ImgOut port is ov**`PORTIDX`**_D |

To any command line you can add a comment, preceded by '#'.

The following code is an example stimuli file which causes the output of an 3x2-image followed by a parameter change (including FID) and output of a second image with dimension 3x1:

```
DAT 00000000 0 0 0 #Format: Cmd Data(hex) EndOfLine EndOfFrame DataValid
DAT 0000001a 0 0 1
DAT 0000001b 0 0 1
DAT 0000001c 0 0 1
DAT 00000000 1 0 1
DAT 0000002a 0 0 1
DAT 0000002b 0 0 1
DAT 0000002c 0 0 1
DAT 00000000 1 1 1
WCK 00000004 0 0 0
FID 00000001 0 0 0
PD0 00000011 0 0 0
```

```
PD1 00000022 0 0 0
PD2 00000033 0 0 0
WCK 00000001 0 0 0
DAT 0000003a 0 0 1
DAT 0000003b 0 0 1
DAT 0000003c 1 1 1
DAT 00000000 0 0 0
DAT 00000000 0 0 0
DAT 00000000 0 0 0
```

The reset and enable ports of the concerning image communication interface are connected to dedicated registers of the slave interface where the affiliation to a process is done according to a parameter **<PORTIDX>_ProcessID**.

### 5.4.1.3. Emulation of ImgIn Interface

The emulation of image communication interfaces of type ImgIn is driven by a stimuli file where information is provided about the sequence of parameter states.

For any present ImgIn port, the eVA_Design_Emulator entity has a generic **<PORTIDX>_StimuliFileName** where **<PORTIDX>** is the name of the corresponding image input port class followed by the port number. The syntax is exactly the same as in the case of the stimuli for ImgOut interfaces except that no **DAT** command is available. A simple stimuli file may look like,

```
WCK 00000010 0 0 0    #Format: Command Data(hex) EndOfLine EndOfFrame DataValid
FID 00000001 0 0 0
PD0 00000111 0 0 0
PD1 00000222 0 0 0
PD2 00000333 0 0 0
WCK 00000001 0 0 0
```

where the parameters **<EndOfLine>**,**<EndOfFrame>** and **<DataValid>** are actually meaningless.

The ImgIn interface emulator writes the received data to file. For that purpose the eVA_Design_Emulator entity has a generic **<PORTIDX>_DumpFileName**. During simulation, a file with the given name is created and the data is written using **DAT** and **WCK** commands in a format which exactly corresponds to the stimuli file format for an ImgOut interface emulator.

The reset and enable ports of the concerning image communication interface are connected to dedicated registers of the slave interface where the affiliation to a process is done according to a parameter **<PORTIDX>_ProcessID**.

### 5.4.1.4. Emulation of Memory Communication

The emulation of memory communication is driven by a stimuli file where information is provided about the sequence of accesses.

For any present memory port, the eVA_Design_Emulator entity has a generic **Mem<X>_StimuliFileName** where **<X>** is the port number. The stimuli file consists of lines with following syntax,

```
<WrData> <WrAddr> <WrFlag> <WrReq> <RdAddr> <RdFlag> <RdReq>
```

where the elements have the following meaning:

**<WrData>**: hexadecimal data word intended for port **ovMemWrDataX**

**<WrAddr>**: hexadecimal write address (for **ovMemWrAddrX**)

**<WrFlag>**: hexadecimal write flag (for **ovMemWrFlagX**)

**<WrReq>**: write request (**oMemWrReqX**)

**<RdAddr>**: hexadecimal read address (for **ovMemRdAddrX**)

**<RdFlag>**: hexadecimal read flag (for **ovMemRdFlagX**)

**<RdReq>**: read request (**oMemRdReqX**)

To any command line, you can add a comment preceded by '#'.

The following code is an example stimuli file which causes the output of two images each consisting of four data words:

```
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
000000000000000a 000000 0 1 000000 00 0 #Write 0xa to address 0x0
000000000000000b 000001 0 1 000000 00 0 #Write 0xb to address 0x1
000000000000000c 000002 0 1 000000 00 0 #Write 0xc to address 0x2
000000000000000d 000003 1 1 000000 00 0 #Write 0xd to address 0x3
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
000000000000000e 000104 1 1 000000 00 0 #Write 0xe to address 0x10a
00000000000000bb 000001 0 1 000000 00 0 #Write 0xbb to address 0x1
0000000000000000 000000 0 0 000000 00 1 #Read from address 0x0
0000000000000000 000000 0 0 000001 00 1 #Read from address 0x1
0000000000000000 000000 0 0 000002 00 1 #Read from address 0x2
0000000000000000 000000 0 0 000003 01 1 #Read from address 0x3
000000000000000e 000000 0 0 000104 00 1 #Read from address 0x104
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
0000000000000000 000000 0 0 000000 00 0
```

Read data returned from the memory to the emulator is written to a file. For providing the file name the eVA_Design_Emulator entity has a generic **Mem<X>_DumpFileName**. The dump lines have the following format,

```
<RdData> <RdFlag>
```

where the elements have the following meaning:

**<RdData>**: hexadecimal data word according to port **ivMemRdDataX**

**<RdFlag>**: hexadecimal value of read flag according to port **ivMemRdFlagX**

### 5.4.1.5. GPIO Emulation

The emulation of dedicated output signals is done for each signal independently, driven by a stimuli file. There information is provided about the sequence of signal states.

The stimuli file may consist of a number of commands which are described below. For any present output signal port the eVA_Design_Emulator entity has a generic **oSig_<NAME>_StimuliFileName** where **<NAME>** is the concerning port name.

The following table describes the available commands:

| Command | Description |
|---|---|
| **SET** | Set signal. This command provides the signal state to which the output at the port oSig_**NAME** |

| Command | Description |
|---|---|
| | will be set. The next command will be executed one clock tick later. It has the syntax<br>`SET <value>`<br><br>where <value> may be 0 or 1. |
| **WCK** | Wait command. It has the syntax<br>`WCK <ticks`<br><br>where the parameter **<ticks>** provides the number of clock ticks for which the signal will be held constant. |
| **RST** | Restart from begin. The command interpreter will start again from the first line of the stimuli file. This command does not have any parameters. The command will execute the first command of the file at the same clock tick allowing assembling a loop without a gap. |
| **STP** | Stop at current state. The command interpreter will stop and the current signal state will be held constant until end of simulation. This command does not have any parameters. |

To any command line you can add a comment, preceded by '**#**'.

The following code is an example stimuli file which causes the output signal toggling being low for 5 clock cycles and high for 7 clock cycles (synchronous to iDesignClk):

```
SET 0              # deassert output
WCK 0004           # wait for 4 clock cycles
SET 1              # assert output
WCK 0006           # wait for 6 clock cycles
RST                # restart from begin
```

Dedicated input signals are monitored writing a dump file **iSig_<NAME>_DumpFileName** where **<NAME>** is the concerning port name. The file is composed of **SET** and **WCK** commands exactly corresponding to the commands of the stimuli file for a dedicated output signal.

### 5.4.2. Embedding the IP Core

To embed the IP core:

1. Integrate all interfaces of the IP core black box into your top-level HDL design. For testing the interfaces as you implement them step by step, use the test bench which has been generated together with the black box (as described in section Section 5.4.1, 'Simulation Framework').

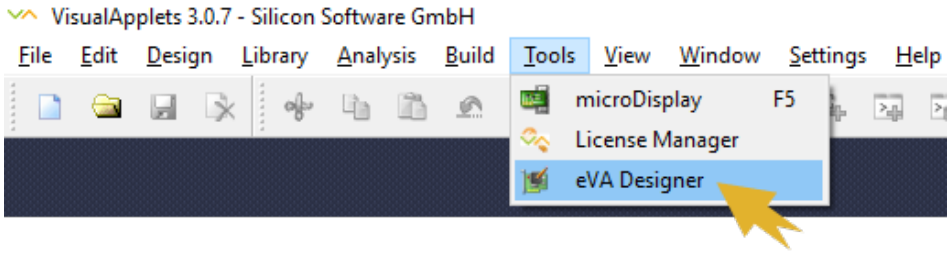
2. After you have fully integrated all black box interfaces into your HDL design, generate a netlist of your design.

3. Create a constraints file.

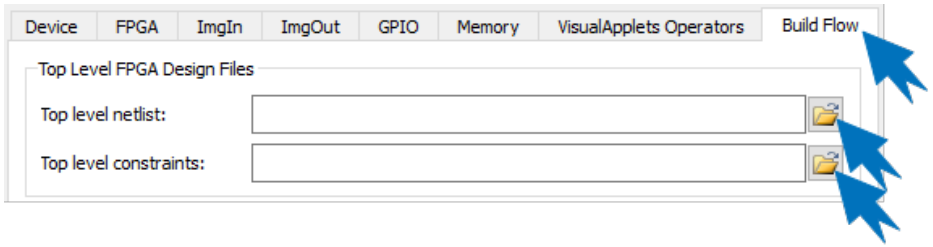
### 5.4.3. Entering Build Flow Details

Before you can actually generate the eVA Plugin that will introduce your HW specific IP core with all its interface specifications to VisualApplets, you need to add some further details to your hardware description file (`*.xml` or `*.eva`) via the GUI of **eVA Designer**:

1. Start VisualApplets and open **eVA Designer**.




2. Go to the **Build Flow** tab.




3. Under **Top Level FPGA Design Files**, enter the paths to the files you created:

   **Top level netlist:** Specify the path to your top level netlist (netlist of your overall FPGA design including wrapped VisualApplets core black box).

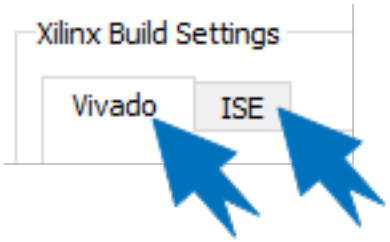
   **Top level constraints:** Specify the path to your user constraints file for your top level netlist (netlist of your overall FPGA design including wrapped VisualApplets core black box). If you use the Xilinx ISE tool chain, this is a `*.ucf` file, if you use the Xilinx Vivado tool chain, this is an `*.xdc` file.

4. Under **Additional FPGA Design Files**, specify the path to these files if required. Such files may be further netlists for additional IP cores with constraint files, etc.



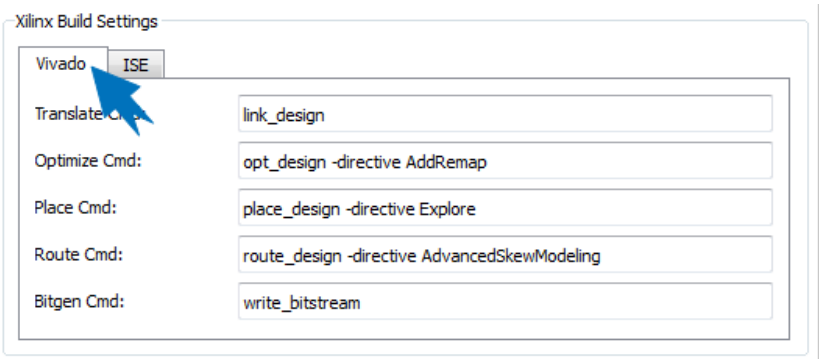

   Under **Xilinx Build Settings**, you now need to specify the tool chain you will be using. Bitstream synthesis requires a 3rd party software: Depending on the used FPGA, this is either Xilinx Vivado WebPACK (free), Xilinx Vivado Design Suite, Xilinx ISE WebPACK (free), or Xilinx ISE Design Suite (registered trade marks of Xilinx Corp.).

5. Select the **Vivado** or **ISE** tab, according to the Xilinx tool chain you use for synthesizing the FPGA bit stream.




6. Enter the details for your tool chain.

   **Vivado**:




If you use Xilinx Vivado, provide the following details:

**Translate Cmd:** Command line for the link_design step of the Vivado implementation flow.

**Optimize Cmd:** Command line for the opt_design step of the Vivado implementation flow.

**Place Cmd:** Command line for the place_design step of the Vivado implementation flow.

**Route Cmd:** Command line for the route_design step of the Vivado implementation flow.

**Bitgen Cmd:** Command line for the write_bitstream step of the Vivado implementation flow.

**ISE**:




If you use Xilinx ISE, provide the following details:

**Translate command:** Command line for the translate step of the ISE implementation flow.

**Map command:** Command line for the map step of the ISE implementation flow.

**Par command:** Command line for the place and route step of the ISE implementation flow.

**Bitgen command:** Command line for the configuration bit stream generation step of the ISE implementation flow.

7. Under **Supplementary Files**, enter additional files you want to have available in the hardware specific directory of VisualApplets if required. Such files might be user documentation, build flow plug-ins, scripts, documents, help tools, etc.



After you have entered all details regarding the build flow, you are ready to generate the eVA plugin installer for your target hardware.

### 5.4.4. Creating the eVA Plugin

The hardware-specific eVA plugin for VisualApplets is created out of the following files:

- Mandatory:
  - Hardware description file (`*.xml`), including operator definitions (see Section 5.3.9, 'Defining Hardware-Specific Operators')
  - Top-level netlist, including wrapped IP core black box (`*.ngc` / `*.edn`) (see box below)
  - Constraints file (`*.ucf` format if you use Xilinx ISE, `*.xdc` format if you use Xilinx Vivado) (see box below)
- Optional:
  - Icon file for the hardware device (`*.png`) for graphical representation of hardware in VisualApplets GUI
  - Icon files for the individual hardware-specific operators (`*.png`) for graphical representation of operators in VisualApplets GUI
  - Help files for the platform specific operators (`*.html`)

All these files you have already entered to **eVA Designer** while working through the steps described in sections Section 5.3, 'Defining the IP Core Properties' and Section 5.4.3, 'Entering Build Flow Details'.



**VisualApplets IP Core Netlist Generation**

You can also create a plugin installer without top-level netlist and constraints file. After installation, you will be able to see how the integration looks like in VisualApplets (available hardware-specific operators etc.).

However, with a plugin that doesn't contain the top-level netlist and constraints file, you cannot build applets that can run on your hardware platform.

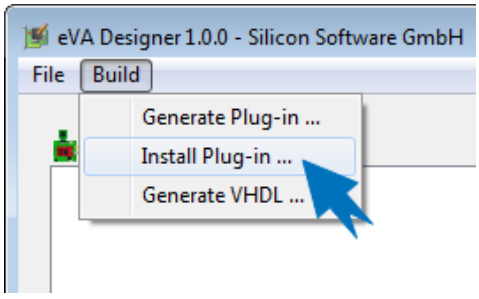
To build the actual eVA Plugin for VisualApplets, you have two options. You can either

- Build and immediately install the eVA Plugin into the VisualApplets installation on your machine.
- Build an executable eVA plugin installer which you can provide to colleagues and/or customers.

#### 5.4.4.1. Direct Installation of Plugin

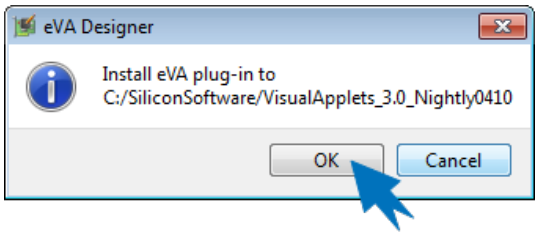
If you want to test or use the plug-in directly on the machine you have been developing it:

1. In **eVA Designer**, open the **Build** menu.

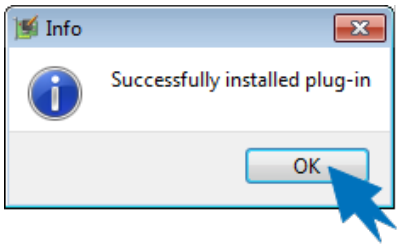


2. Select menu item **Install Plug-in ...**

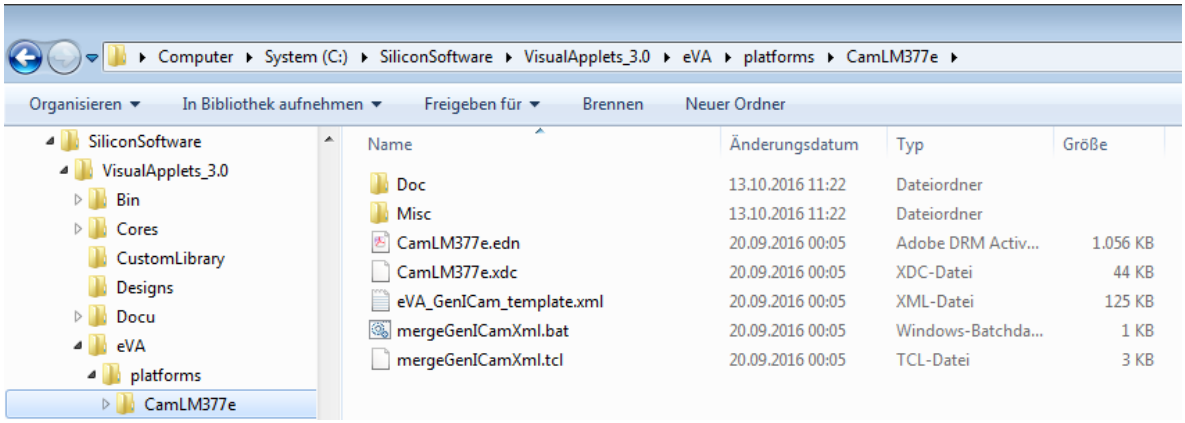
3. Enter the path to your VisualApplets installation directory, or confirm the suggested installation directory.




You are informed about the successful installation:



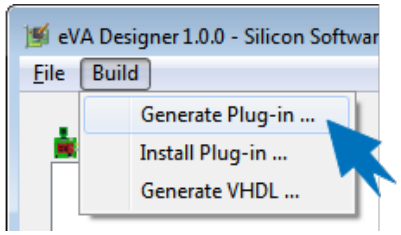

The files of the plug-in you find in the VisualApplets installation directory, subdirectory eVA/`platforms/<devicename>`:




## 5.4.4.2. Building the eVA Plugin Installer

To create an executable that you can forward to VisualApplets developers that will design image processing applications for your hardware:

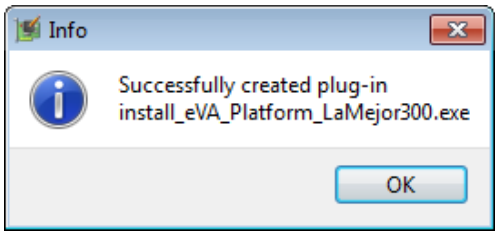
1. In **eVA Designer**, open the **Build** menu.



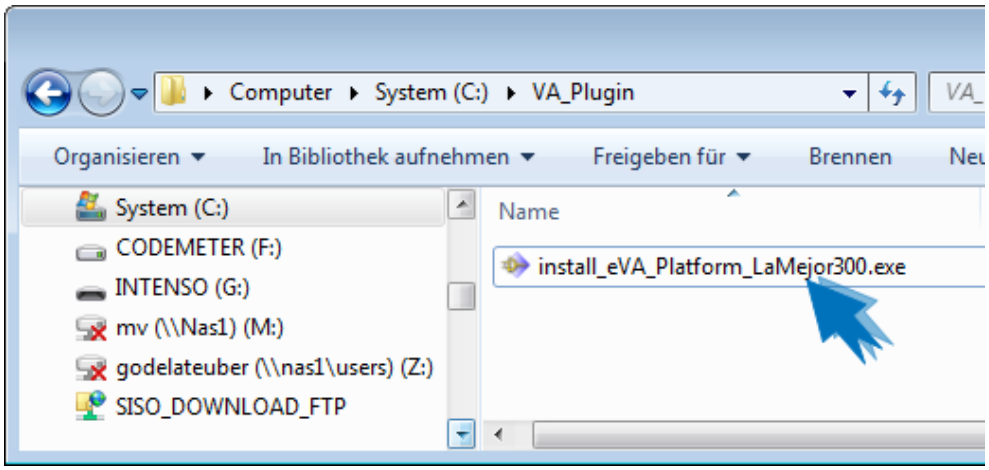

2. Select menu item **Generate Plug-in ...**

3. Select the target folder where you want to save the plugin.

As soon as you select the target folder, the eVA Plugin is created immediately. You are informed via message that your build was successful:




The plugin is available in the specified folder:

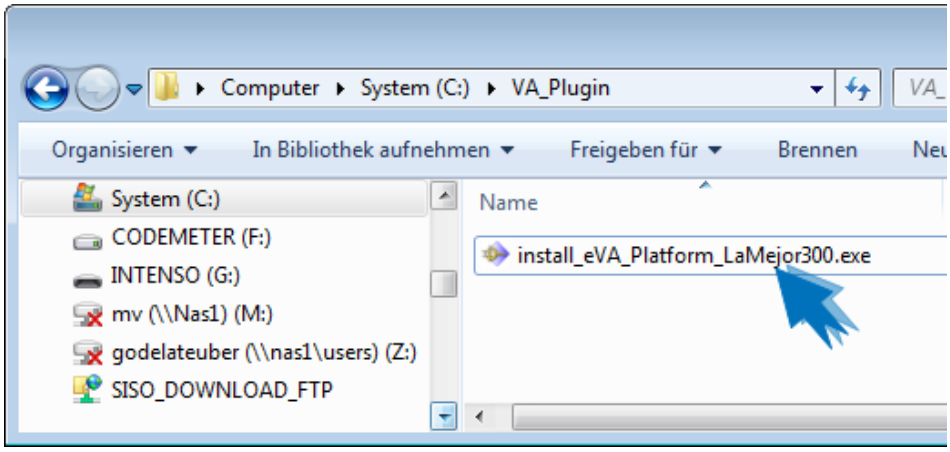



You can now distribute the plugin.

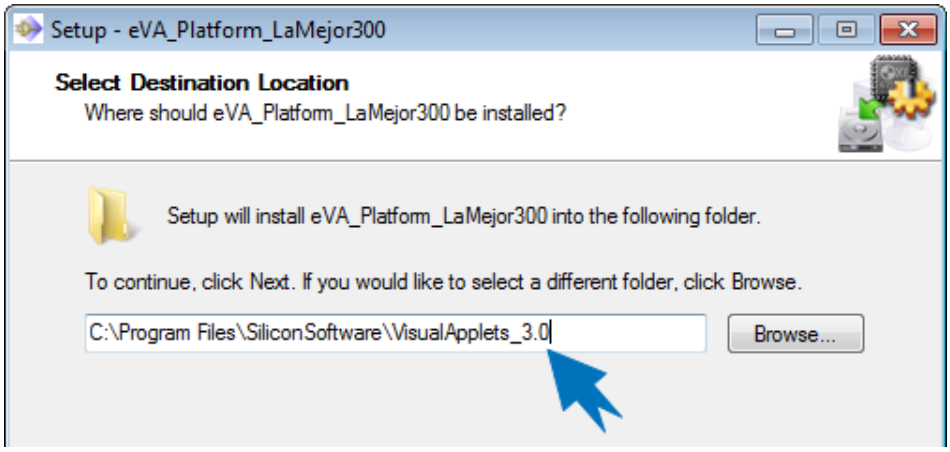
### 5.4.4.3. Executing the eVA Plugin Installer

To install the eVA Plugin into an existing VisualApplets installation:

1. Close your VisualApplets installation.
2. Double-click the plugin `*.exe` file in your file system.




3. Follow the instructions of the installer. The installer automatically suggests the installation directory of VisualApplets:



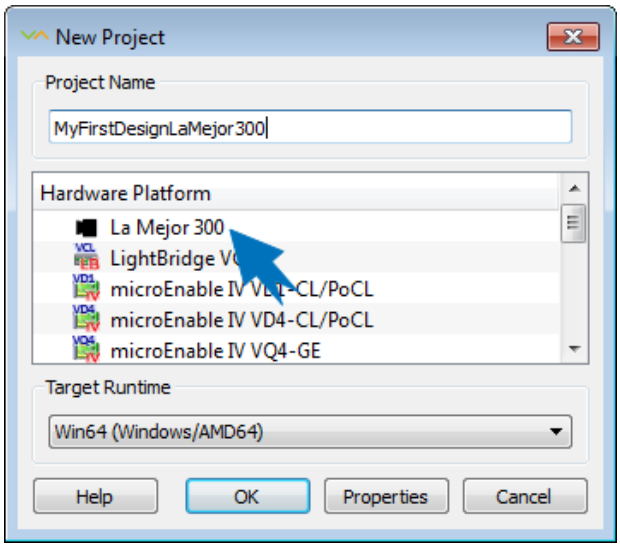

4. Click **Finish** to complete the installation process.

### 5.4.5. Using the Installed eVA Plugin

After installation, you can design new image processing applications for your platform in VisualApplets:

1. Open VisualApplets.
2. From menu File, select **New**.

Now, you can select your hardware platform and start designing in VisualApplets:




For information how to design image acquisition and processing applications with VisualApplets, consult the comprehensive VisualApplets online Documentation at 2. *Getting Started*.

## 5.5. Runtime Software Interface

VisualApplets supports different models for accessing design parameters at runtime:

- Using eVA runtime environment based on HAP files
- Using GenICam API based on generated GenICam XML code
- Using generic, design-specific C API code generated by VisualApplets

### 5.5.1. HAP-Based eVA Runtime Interface

VisualApplets controls synthesis and implementation of the FPGA design. The resulting design is saved in a proprietary Basler file format (*.hap). This format contains the configuration bitstream for the FPGA and means for realizing a design-specific software interface.

In general, VisualApplets operators contain dynamic parameters which can be modified during runtime. The HAP file contains all necessary information for accessing design specific parameters easily.

For proper function of the runtime software on the user platform, the following requirements must be fulfilled:

- The user needs access to the FPGA bitstream as he needs to do the configuration. For that purpose, the runtime interface provides the function `vaRt_GetConfig()`.
- For accessing the slave interface for the design registers, the user must implement the functions `WriteReg32()`, `ReadReg32()`, or optionally `WriteReg64()` and `ReadReg64()`. The runtime software is notified about these functions when the function `vaRt_InitParams()` is called.

Then the VisualApplets bitstream can be configured and the runtime software can write and read design parameters.

The register access functions must comply with following function types:

```
typedef int ReadReg32(void* device, const uintptr_t address, uint32_t *value)
typedef int WriteReg32(void* device, const uintptr_t address, uint32_t value)
typedef int ReadReg64(void* device, const uintptr_t address, uint64_t *value)
typedef int WriteReg64(void* device, const uintptr_t address, uint64_t value)
```

The software interface library provides additional functions for controlling the VisualApplets design. In the following table, the full set of access functions is listed:

| Nr. | Function | Description |
|---|---|---|
| 1 | `vaRt_OpenHap()` | Opens the hardware applet `*.hap` and returns a pointer to the HAP structure which serves as a handle for further accesses. |
| 2 | `vaRt_CloseHap()` | Closes the hardware applet which has been opened by `vaRt_OpenHap`. |
| 3 | `vaRt_GetConfig()` | Returns a pointer to the configuration bit stream for the FPGA. |
| 4 | `vaRt_GetAppletProperty()` | Query function for applet properties like the maximum frequency of the clock net *iDesignClk* of the applet, information about the target platform, and version information. |
| 5 | `vaRt_GetProcessCount()` | Returns the number of processes of which the design is composed. |
| 6 | `vaRt_GetIoPortCount()` | Returns the number of used external interface ports. All kinds of interface ports are joined in a list where the |

| Nr. | Function | Description |
|---|---|---|
| | | position in the list is the logical number of the port. Port names and properties can be obtained by below specified functions `vaRt_GetIoPort…`. |
| 7 | `vaRt_GetIoPortName()` | Get the name of the interface port with the given logical port number. |
| 8 | `vaRt_GetIoPortProperty()` | Query I/O port properties like the name of a connected operator, the process which controls this port, and the current image format setting for this port. |
| 9 | `vaRt_GetParamCount()` | Returns the number of parameters of a design. |
| 10 | `vaRt_GetParamName()` | For a given parameter number the corresponding name is returned. |
| 11 | `vaRt_GetParamId()` | For a given parameter number the parameter ID is returned. |
| 12 | `vaRt_GetParamIdByName()` | For a given parameter name the parameter ID is returned. |
| 13 | `vaRt_GetParamProperty()` | Query parameter properties. |
| 14 | `vaRt_InitParams()` | Configure parameter interface and initialize parameters to default values. |
| 15 | `vaRt_GetParam()` | Query parameter value. |
| 16 | `vaRt_GetParamArray()` | Query array of parameters. |
| 17 | `vaRt_SetParam()` | Set parameter value. |
| 18 | `vaRt_SetParamArray()` | Set array of parameters. |
| 19 | `vaRt_SetGlobalEnable()` | Set the *Master Enable* signal. |
| 20 | `vaRt_SetProcessEnable()` | Activate a process of the VisualApplets design. |
| 21 | `vaRt_ResetProcess()` | Perform a reset of a VisualApplets design process. |
| 22 | `vaRt_GetErrorDescription()` | Query description for given error code. |

### 5.5.1.1. Communicating Data

For querying information and configuring parameters, data must be exchanged through the runtime interface. In order to keep the interface functions simple but providing a type-save interface, an abstraction mechanism for data is implemented. Whenever data of different types needs to be communicated, a data structure called `va_data` is used, containing a reference to the data and information about the underlying data type. This data structure is created by the user but configured by dedicated functions listed below. The following table shows the data types which are handled by this method:

| Data Type | Description |
|---|---|
| `VA_ENUM` | Enum entry given as 32-bit integer. |
| `VA_INT32` | 32-bit signed integer. |

| Data Type | Description |
|---|---|
| VA_UINT32 | 32-bit unsigned integer. |
| VA_INT64 | 64-bit signed integer. |
| VA_UINT64 | 64-bit unsigned integer. |
| VA_DOUBLE | Floating-point value, double precision. |
| VA_INT32_ARRAY | Array of 32-bit signed integer numbers. |
| VA_UINT32_ARRAY | Array of 32-bit unsigned integer numbers. |
| VA_INT64_ARRAY | Array of 64-bit signed integer numbers. |
| VA_UINT64_ARRAY | Array of 64-bit unsigned integer numbers. |
| VA_DOUBLE_ARRAY | Array of double numbers. |
| VA_STRING | String given as const char*. |

Configuring an earlier created va_data structure (vaData) for setting up data communication is done via the following functions:

```
va_data* va_data_enum(va_data* vaData, int32_t *data)
va_data* va_data_int32(va_data* vaData, int32_t *data)
va_data* va_data_uint32(va_data* vaData, uint32_t *data)
va_data* va_data_int64(va_data* vaData, int64_t *data)
va_data* va_data_uint64(va_data* vaData, uint64_t *data)
va_data* va_data_double(va_data* vaData, double *data)
va_data* va_data_int32_array(va_data* vaData, int32_t *data, size_t elementCount)
va_data* va_data_uint32_array (va_data* vaData, uint32_t *data, size_t elementCount)
va_data* va_data_int64_array (va_data* vaData, int64_t *data, size_t elementCount)
va_data* va_data_uint64_array (va_data* vaData, uint64_t *data, size_t elementCount)
va_data* va_data_double_array (va_data* vaData, double *data, size_t elementCount)
va_data* va_data_string(va_data* vaData, char *data, size_t strSize)
va_data* va_data_const_string(va_data* vaData, const char **data)
```

For strings there are two options how strings are communicated:

- Providing a char array via va_data_string(). Then queried string data will be copied to that array.
- Providing a pointer to const char*. Then a pointer to an internal string representation is returned when information of type VA_STRING is queried. When you use this approach check the live time of the returned string.

Example Code:

The following example shows code for querying the design frequency for a configured design.

```
double desFreq;
va_data va_desFreq;
va_data_double(&amp;va_desFreq,&amp;desFreq);

vaRt_GetAppletProperty(hapHandle, &quot;DesignFreq&quot;,&amp;va_desFreq);
```

After that, the variable desFreq will contain the requested information.

### 5.5.1.2. Detailed Description of Interface Functions

The following table gives a detailed description of parameters and returned values for the specified runtime access functions.

| **Function** | **int vaRt_OpenHap (const char* hapFileName, va_hap_handle*hap)** |
|---|---|
| Parameter 1 | Name of hap file which shall be opened. |
| Parameter 2 | Return pointer for hap handle. |
| Description | Opens the hardware applet with the provided name and returns a pointer to the hap structure.<br><br>The returned pointer is used as a handle in all other interface functions. |
| Return value | 0 : OK<br><br><0: Error opening applet |

| **Function** | **int vaRt_CloseHap (va_hap_handle handle)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Description | Closes the VA hardware applet. |
| Return value | 0 = OK<br><br><0 = Error closing HAP |

| **Function** | **int vaRt_GetConfig (va_hap_handle handle, int fpgaID, char**data, size_t *sizeInBytes)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | ID of the FPGA within the target platform. For single-configuration HAP files (currently the only option) this parameter may be fixed to -1. Otherwise this parameter must correspond to the (optional) FPGA ID specified in the hardware definition file used for creating the VisualApplets platform. |
| Parameter 3 | Pointer to buffer for the configuration bitstream. |
| Parameter 4 | Communicate the size of the configuration bitstream in bytes. |
| Description | Get the configuration bitstream for the FPGA design. The function has two modes of operation: |
| Return value | 0: OK<br><br><0: Error acquiring the bitstream |

| **Function** | **int vaRt_GetAppletProperty (va_hap_handle handle, const char* propName,va_data * data)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Name of property which shall be queried. |
| Parameter 3 | Pointer to data structure which will used for communication. |
| Description | Get property of the applet. Following properties may be queried:<br><br>• V*isualAppletsVersion*: Get version of VisualApplets framework used for creating this applet (type VA_STRING) |

| **Function** | **int vaRt_GetAppletProperty (va_hap_handle handle, const char* propName,va_data * data)** |
|---|---|
| | • *IPCoreVersion*: Get version of VA IP Core (type VA_STRING)<br>• *AppletVersion*: Get applet version (type VA_STRING)<br>• *DesignFreq*: Get frequency of design clock in MHz (type VA_DOUBLE)<br>• *PlatformVendor*: Vendor of platform (type VA_STRING)<br>• *PlatformType*: Name of hardware platform (type VA_STRING)<br>• *PlatformID*: Identification number of platform device (type VA_STRING)<br>• *PlatformVersion*: Platform device version as defined in the hardware description used for creating the applet (type VA_STRING)<br>The properties are identified by one of the above strings (provided through parameter 2) and communicated via the data structure given through parameter 3. For properties of type VA_STRING the returned string may be overwritten by the next call of this function. |
| Return value | 0 : OK<br><0 : Can’t retrieve applet property |

| **Function** | **int vaRt_GetProcessCount (va_hap_handle handle, unsigned int* count)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Return value for process count. |
| Description | Get the number of processes of which the design is composed. |
| Return value | 0: OK<br><0 : Can’t retrieve information |

| **Function** | **int vaRt_GetIoPortCount (va_hap_handle handle, unsigned int*count)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Return value for port count. |
| Description | Get the number of used I/O ports of the VA IP Core. The returned number represents the size of an internally managed port list where all interface ports are registered and the position in the list is the logical number of the I/O port. |
| Return value | 0: OK<br><0 : Can’t retrieve information |

| **Function** | **int vaRt_GetIoPortName (va_hap_handle handle, unsigned int index, const char** name)** |
|---|---|
| Parameter 1 | Hap handle from `vaRt_OpenHap()`. |
| Parameter 2 | Logical interface port number, which is an integer between Zero and the returned value from `vaRt_GetIoPortCount()` minus 1. |
| Description | Get the name of the interface port with the given logical number. When there are several ports of the same kind these ports are differentiated by a suffix **_X** where **X** is the port index (e.g. *DmaRd_0*). |
| Return value | 0: OK; `*name` returns a pointer to a static buffer which holds the requested name (0-terminated). Note that the buffer may be overwritten by the next call of this function<br><br><0 : No port with given index found |

| **Function** | **int vaRt_GetIoPortProperty (va_hap_handle handle, const char* portName, const char* propName, va_data* data)** |
|---|---|
| Parameter 1 | Hap handle from `vaRt_OpenHap()`. |
| Parameter 2 | Name of I/O port. |
| Parameter 3 | Name of property which shall be queried. |
| Parameter 4 | Pointer to data structure which will be used for communication. |
| Description | Get property of the I/O port with the given name. Following properties may be queried:<br><br>• *OperatorName*: User defined name of connected operator (type `VA_STRING`).<br><br>• *OperatorType*: Operator type name of connected operator (type `VA_STRING`).<br><br>• *Process*: Logical number of process in which the connected operator works (type `VA_UINT32`).<br><br>• *FormatID*: Format ID of I/O operator (type `VA_UINT32`, only available for ports of type *ImgIn* or I*mgOut*).<br><br>The properties are identified by one of the above strings (provided through parameter 3) and communicated via the data structure given through parameter 4. For properties of type `VA_STRING` the returned string may be overwritten by the next call of this function. |
| Return value | 0: OK<br><br><0 : Can’t retrieve the requested property |

| **Function** | **int vaRt_InitParams (va_hap_handle handle, struct va_device *device, ReadReg32 *read32, WriteReg32 *write32, ReadReg64 *read64, WriteReg64 *write64)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Pointer to device handle which is provided to the low level access functions WriteReg32(), ReadReg32(), WriteReg64, and ReadReg64() for any register access. |
| Parameter 3 | Function pointer for 32-bit register read function. |
| Parameter 4 | Function pointer for 32-bit register write function. |
| Parameter 5 | Function pointer for 64-bit register read function. |
| Parameter 6 | Function pointer for 64-bit register write function. |
| Description | Initialize parameter interface by providing low level register access functions and associating a board interface pointer to them. The initialization procedure includes configuring initial values. Note that any other function vaRt_*Param* and the functions for controlling reset and enable may only be called after this function has been executed successfully.<br><br>Regarding the register access functions following options are available: |
| Return value | 0: Parameter interface has been initialized successfully<br><br><0 : Error during initialization |

| **Function** | **int vaRt_GetParamCount (va_hap_handle handle, unsigned int*count)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Return value for number of parameters. |
| Description | Get the number of parameters of the applet which can be accessed via the parameter interface. The returned number represents the size of an internally managed parameter list where all available parameters are registered and the position in the list defines the logical number of the parameter. |
| Return value | 0: OK<br><br><0 : Can’t retrieve information |

| **Function** | **int vaRt_GetParamName (va_hap_handle handle, unsigned int index, const char** name)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Logical parameter number, which is an integer between Zero and the returned value from vaRt_GetParamCount() minus 1. |

| **Function** | **int vaRt_GetParamName (va_hap_handle handle, unsigned int index, const char** name)** |
|---|---|
| Parameter 3 | Return pointer to name string. |
| Description | Get the name of the applet parameter by the given logical parameter number. |
| Return value | 0: OK; *name returns a pointer to a static buffer which holds the parameter name (0-terminated). Note that the buffer may be overwritten by the next call of this function.<br><br><0 : Can't retrieve information |

| **Function** | **int vaRt_GetParamId (const struct va_hap_handle* handle, unsigned int index, int* paramId)** |
|---|---|
| Parameter 1 | Hap handle from `vaRt_OpenHap()`. |
| Parameter 2 | Logical parameter number, which is an integer between zero and the returned value from `vaRt_GetParamCount()` minus 1. |
| Parameter 3 | Return pointer for parameter ID. |
| Description | Get the ID of a parameter of the applet. This ID is used for accessing that parameter. |
| Return value | 0: OK<br><br><0 : No parameter with given index found |

| **Function** | **int vaRt_GetParamIdByName (const struct va_hap_handle* handle, const char* name, int* paramId)** |
|---|---|
| Parameter 1 | Hap handle from `vaRt_OpenHap()`. |
| Parameter 2 | Name of parameter which can be one of the values provided by `vaRt_GetParamName()`. |
| Parameter 3 | Return pointer for parameter ID. |
| Description | Get the ID of a parameter with the given name. This ID is used for accessing that parameter. |
| Return value | 0: OK<br><br><0 : No parameter with given name found |

| **Function** | **int vaRt_GetParamProperty (va_hap_handle handle, int paramID, const char* propName, va_data* data)** |
|---|---|
| Parameter 1 | Hap handle from `vaRt_OpenHap()`. |
| Parameter 2 | Parameter name. |
| Parameter 3 | Name of property which shall be queried. |
| Parameter 4 | Pointer to data structure which will be used for communication. |

| **Function** | **int vaRt_GetParamProperty (va_hap_handle handle, int paramID, const char* propName, va_data* data)** |
|---|---|
| Description | Query property of applet parameter. Different parameter types do have different properties. See the VA Engine specification for a list of available parameter properties. The properties are identified by a string given by parameter 3 and communicated via the data structure provided by parameter 4. For properties of type VA_STRING the returned string is stable at least until the parameter gets modified. |
| Return value | 0 : OK<br><br><0 : Can’t retrieve parameter property |

| **Function** | **int vaRt_GetParam (va_hap_handle handle, int paramID, va_data * value)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Parameter ID. |
| Parameter 3 | Pointer to data structure which will be used for communication. The associated data element will be overwritten by the acquired value. |
| Description | Query the applet parameter by parameter ID. The data is communicated using a data structure given by parameter 3. For querying field parameters use vaRt_GetParamArray(). For properties of type VA_STRING the returned string is stable at least until the parameter gets modified. |
| Return value | 0: OK.<br><br><0 : Error querying the parameter. |

| **Function** | **int vaRt_GetParamArray (va_hap_handle handle, int paramID, va_data* values, size_t startIndex, size_t elementCount)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Parameter ID. |
| Parameter 3 | Pointer to the data structure which will be used for communication. The associated data elements will be overwritten by the acquired values. |
| Parameter 4 | Start index within the parameter field |
| Parameter 5 | Number of elements which shall be queried |
| Description | Query field parameter of applet by parameter ID filling an array of data elements. The data is communicated via the data structure given by parameter 4. |
| Return value | 0: OK.<br><br><0 : Error querying the array. |

| **Function** | **int vaRt_SetParam (va_hap_handle handle, int paramID, const va_data * value)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Parameter ID. |
| Parameter 3 | Pointer to the data structure which will be used for submitting data to the Applet. |
| Description | Set applet parameter with the given name. Data is provided by a data structure (Parameter 4). For setting field parameters use vaRt_SetParamArray(). |
| Return value | 0: OK.<br><br><0 : Error setting the parameter. |

| **Function** | **int vaRt_SetParamArray (va_hap_handle handle, int paramID, const va_data * values, size_t startIndex, size_t elementCount)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | Parameter ID. |
| Parameter 3 | Pointer to the data structure which will be used for submitting values to the applet. |
| Parameter 4 | Start index within parameter field. |
| Parameter 5 | Number of elements which shall be set. |
| Description | Set field parameter of applet with the given name submitting an array of data elements. The data is provided via a data structure (Parameter 3). |
| Return value | 0: OK.<br><br><0 : Error setting the parameter field. |

| **Function** | **int vaRt_SetGlobalEnable (va_hap_handle handle, int active)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |
| Parameter 2 | State to which the global enable shall be set (1 or 0). |
| Description | Set the global enable to 1 (active=1) or 0 (active=0). |
| Return value | 0: OK<br><br><0 : Error setting the global enable state. |

| **Function** | **int vaRt_SetProcessEnable (va_hap_handle handle, unsigned int procNr, int active)** |
|---|---|
| Parameter 1 | Hap handle from vaRt_OpenHap(). |

| **Function** | **int vaRt_SetProcessEnable (va_hap_handle handle, unsigned int procNr, int active)** |
|---|---|
| Parameter 2 | Process number or ((unsigned int)-1) for identifying all processes at once. |
| Parameter 3 | State to which the enable shall be set (1 or 0). |
| Description | Set the enable of a process (or all processes) to 1 (active=1) or 0 (active=0). |
| Return value | 0: OK.<br><br><0 : Error setting the process enable state. |

| **Function** | **int vaRt_ResetProcess (va_hap_handle handle, unsigned int procNr)** |
|---|---|
| Parameter 1 | Hap handle from `vaRt_OpenHap()`. |
| Parameter 2 | Process number or ((unsigned int)-1) for identifying all processes at once. |
| Description | Reset the process with the given logical process number (or all processes). The according process reset signal(s) will be asserted and released again. The enable signal(s) of the corresponding process(es) is not touched. Use `vaRt_SetProcessEnable()` and this function for controlling the applet. |
| Return value | 0: OK.<br><br><0 : Cannot reset the process. |

| **Function** | **const char* vaRt_GetErrorDescription (va_hap_handle handle, int errorCode)** |
|---|---|
| Parameter 1 | Hap handle from `vaRt_OpenHap()`. |
| Parameter2 | Code from `vaRt_GetLastError()`. |
| Description | Query error message string for given error code. |
| Return value | Not NULL: Error description (0-terminated c-string)<br><br>NULL: No description available |

### 5.5.2. Runtime Interface Based on GenICam API Version 2.0

For a given design, VisualApplets may generate GenICam XML code. When the target platform is connected via a GenICam compatible interface to the software, this option allows a seamless integration of image processing parameters into the GenICam API. There is no need for any additional software component.

If the parameter access needs to be integrated into an existing master GenICam XML file, you need to do some post processing. This is usually the case when the platform is already controlled via GenICam before embedding the VA IP core.

To integrate the parameter access into an existing master GenICam XML file:

1. Cut the header of the XML file generated by VisualApplets.
2. Copy the remaining code for the categories, variables, and registers into the master XML file.

VisualApplets may automatize this step as it is capable calling custom post processing tasks.

### 5.5.3. Runtime Interface Based on Generated API Code

For a given design, VisualApplets may generate generic C API code. The code is platform independent ANSI-C code. This approach is well suited for software integration in embedded systems (e.g., Zynq7000).

The generated code provides the following interface:

```
#include <stdint.h>

typedef int (*write_func_t)(void* boardHandle,int address, uint64_t value, size_t sizeInBytes);
int (*read_func_t)(void* boardHandle,int address, uint64_t *value, size_t sizeInBytes);

enum VariableTypes_enum{

TYPE_UNKNOWN = 0,

TYPE_INT = 1,

TYPE_FLOAT = 2,

TYPE_STRING = 3

};

typedef enum VariableTypes_enum VariableType_t;

enum AccessMode_enum{
RO,
WO,
RW
};

typedef enum AccessMode_enum AccessMode_t;

int va_init(void* boardHandle, write_func_t wrFunc, read_func_t rdFunc);

int va_set_property(const char* propName, const char* propValue);

int va_get_int_value(const char* varName, int64_t *retValue);

int va_set_int_value(const char* varName, int64_t value);

int va_get_float_value(const char* varName, double *retValue);

int va_set_float_value(const char* varName, double value);

int va_get_string_value(const char* varName, const char **retValue);

int va_get_enum_value(const char* varName, const char** retValue);

int va_set_enum_value(const char* varName, const char* value);

int va_get_enum_item_count(const char* varName, int64_t *itemCount);

int va_get_enum_item_info(const char* varName, int64_t index, const char** itemName, int64_t *itemValue);

int va_query_variables_count(int32_t *retCount);

int va_query_variable_name(int32_t index, const char** retName);

int va_query_variable_type(const char* varName, VariableType_t *type);

int va_query_int_variable_properties(const char* varName,

int64_t *from,

int64_t *to,

int64_t *inc,

AccessMode_t *access);
```

```
int va_query_float_variable_properties(const char* varName,
double *from,
double *to,
double *inc,
AccessMode_t *access);
```

You initialize the interface by calling `va_init()`.

The callback functions for write and read access to the register slave interface of the VA IP core are registered. The given `boardHandle` is stored and provided with any triggered call of these write and read functions. The set of query functions provide a baseline mechanism for implementing a generic runtime interface. For a given design the set functions and get functions provide parameter access where parameters are addressed by name. These functions perform the translation from accessing design parameters to accessing registers on the hardware via the call back functions for register access.

## 5.6. Licensing Model

Each hardware device that uses an VA IP Core is equipped with one runtime license. The costs of a runtime license are defined by the maximum sensor bandwidth, the number of FPGA resources that can be maximally used by the image processing application, and by the DRAM interface availability.

You can generate the runtime licenses yourself. For generating licenses, you get a dongle from Basler that works on a pay-per-use basis. On the dongle, a predefined number of utilization units (1 unit per runtime license) for a specific performance class is provided. When all units on the dongle have been consumed, you can buy a new allotment of units which is loaded onto the dongle.

Your advantages of pay-per-use:

1. No additional hardware on your device required
2. Reliable protection of your intellectual property as the FPGA-internal serial number (DNA of FPGA) is incorporated into the license
3. Easy integration of licensing into the production flow

You can select from the following performance classes:

### 5.6.1. Economy

Moderate Resources, no DRAM Interfaces

| **Bandwidth Classes** | **150 MB/s** | **500 MB/s** | **1000 MB/s** | **unlimited** |
|---|---|---|---|---|
| Performance Class | E150 | E500 | E1000 | **EL** |
| Sensor Bandwidth | Up to 150 MB/s | Up to 500 MB/s | Up to 1000 MB/s | unlimited |
| Available Resources | Up to 50.000 LUT4 | Up to 100.000 LUT4 | Up to 150.000 LUT4 | Up to 200.000 LUT4 |
| DRAM Interfaces | Not supported | Not supported | Not supported | Not supported |

### 5.6.2. eXtended

High Resources and Support of DRAM

| Performance Class | X150 | X500 | X1000 | XL |
|---|---|---|---|---|
| Sensor Bandwidth | Up to 150 MB/s | Up to 500 MB/s | Up to 1000 MB/s | unlimited |
| Available Resources | Up to 100.000 LUT4 | Up to 300.000 LUT4 | Up to 500.000 LUT4 | Up to 750.000 LUT4 |
| DRAM Interfaces | Supported | Supported | Supported | Supported |

### 5.6.3. Superior

Maximum Resources and Support of DRAM

| Performance Class | S150 | S500 | S1000 | SL |
|---|---|---|---|---|
| Sensor Bandwidth | Up to 150 MB/s | Up to 500 MB/s | Up to 1000 MB/s | unlimited |
| Available Resources | unlimited | unlimited | unlimited | unlimited |
| DRAM Interfaces | Supported | Supported | Supported | Supported |

## 5.7. Application Notes

### 5.7.1. Designing for Non-Stoppable Image Sources

When the incoming stream of image data is not stoppable, as it is typically the case when data comes from a camera sensor, special care must be taken in order to avoid overflow of input buffers. Consider the following effects:

- M-type operators may slow down processing (i.e., *PARALLELdn*).
- Synchronizing different data streams with *SYNC* may introduce wait cycles.
- Some kernel based operators like *FIRkernelNxM* require extra clock cycles before accepting new data after the end of a line and even more after the end of a frame.

In general, it is a good idea to instrument the input data path with a circuit like the following one for monitoring whether a design can deal with the input data rate:

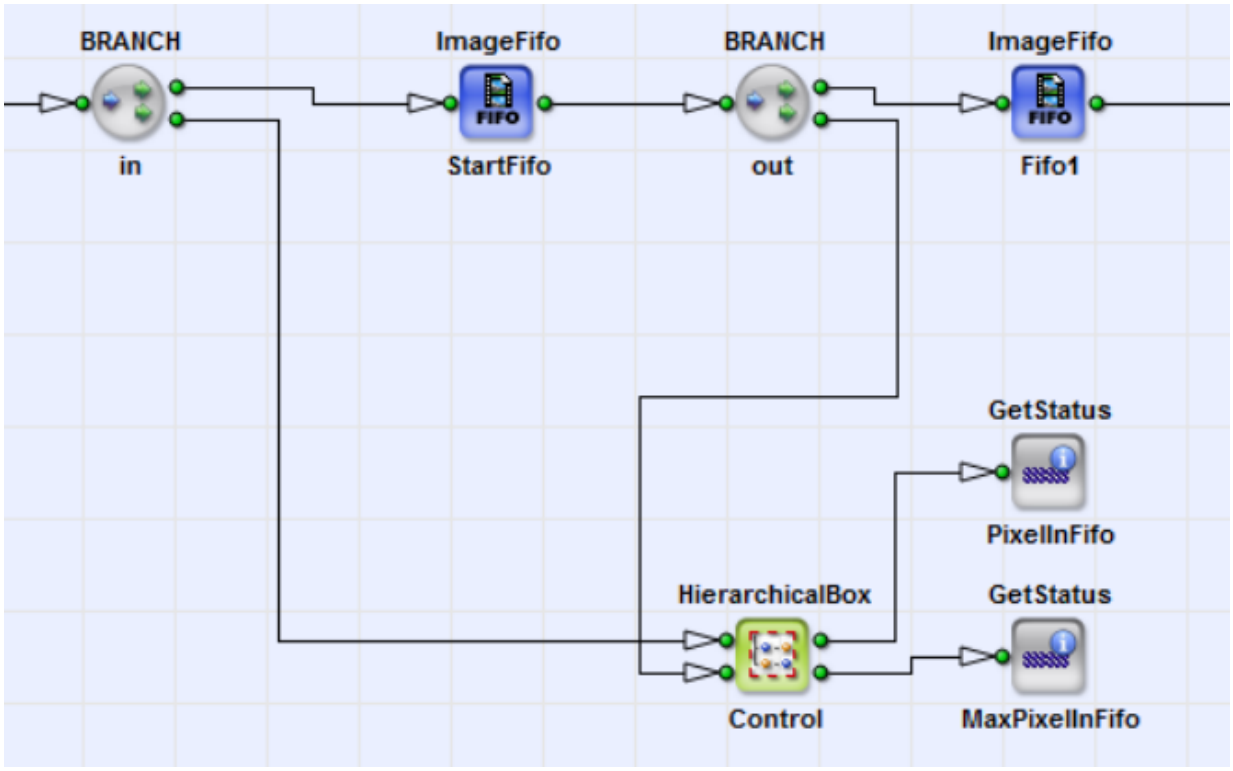



Figure 5.19. Circuit for Monitoring the Input Data Rate

The hierarchical box *Control* is designed as follows:

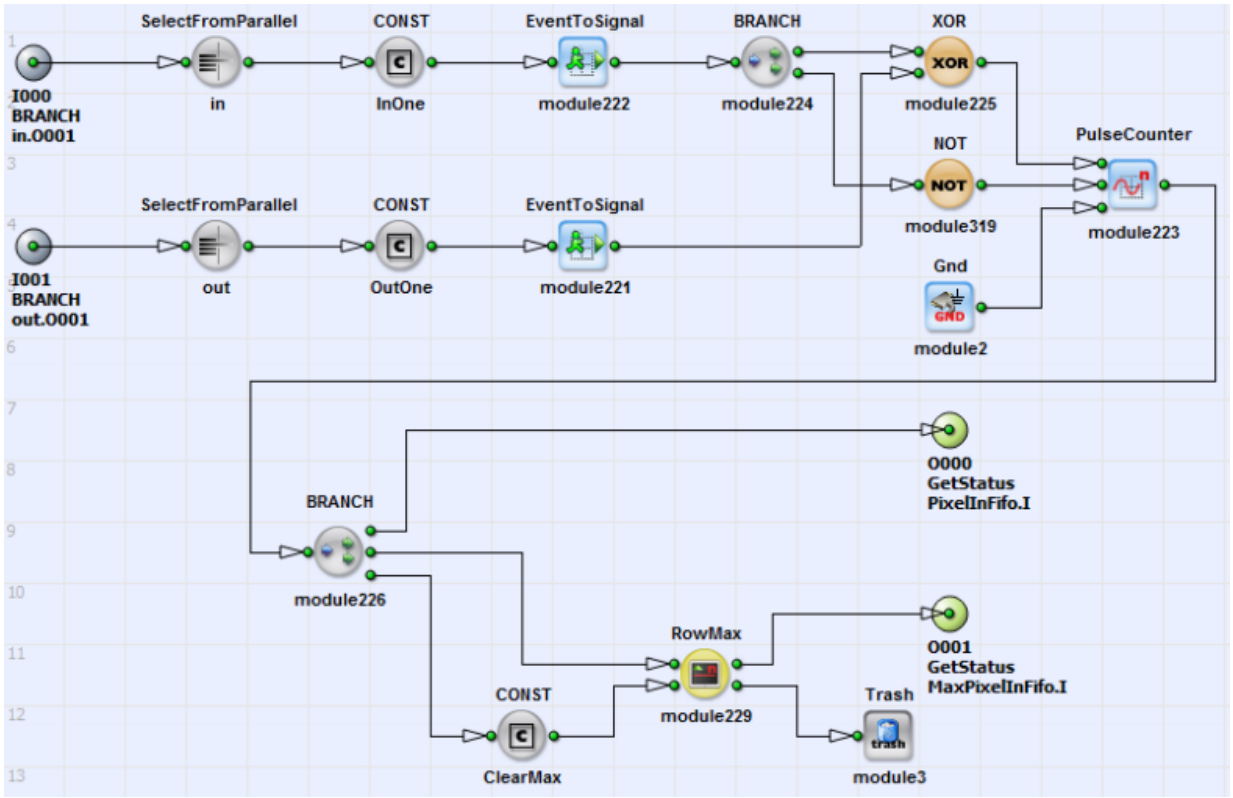



Figure 5.20. Control Hierarchical Box

This circuit measures the number of buffered data words between the input and the output of FIFO *StartFifo*. The FIFO *StartFifo* is configured for taking a non-stoppable data stream (parameter *InfiniteSource* = ENABLED). It is useful to define a buffer size so one or a few lines can be stored in that FIFO.

Check the read parameter of the operator *MaxPixelInFifo* during or after operation. If the subsequent image processing pipeline is able to deal with the input data rate there should be a low number showing that the FIFO is only needed to buffer a few data words.

If you find that your design cannot operate with the input data rate, you have following options:

- Increase the throughput of your design, in particular by increasing the parallelism.
- Decrease the amount of data which needs to be processed, for example, you may operate on a reduced image size by dropping some data from the sensor.
- Use a RAM-based image buffer operator for decoupling the input data rate from the processing rate.

## 5.7.2. GenICam API

Some VisualApplets operators require that their parameters are set before the processing is started. Other operators require that parameters are not changed during processing.

To avoid problems with parameterizing a design, it is generally helpful to set all parameters which don't need to be changed at runtime in your VisualApplets design to *Static*. If your applet is not working as expected, try to write all dynamical parameters before you start processing.

If you need to change an applet parameter during processing check the operator documentation whether this is supported.

### 5.7.2.1. Operators Requiring Startup Initialization

*SplitLine*

If you are working with VisualApplets version 3.0.4 or lower:

The operator *SplitLine* has a parameter *LineLength* which may be dynamic or static. If you want to use parameter *LineLength* as a dynamic parameter, the parameter value must be written before the processing is started.

### 5.7.2.2. Operator Parameters which may not be Changed During Running

The parametrization of some operators can be updated during processing only to a limited extend. These are the following operators:

*ImageBufferMultiRoI*

In some operation modes this operator ignores parameters which are set during processing. In these cases the parent process must be stopped before writing the parameters. For details, see *ImageBufferMultiRoI*.

*SelectROI*

Change the parameter settings of operator *SelectROI* only after the image data flow has been terminated, i.e., when no image data entered the operator after the last end of frame.

## 5.7.3. Deviating Parameter Interface During Runtime

On platforms using embedded VisualApplets (eVA), during runtime some VisualApplets operators offer a parameter interface that slightly differs from the list of parameters which is presented in the VisualApplets GUI. This is true for the following nine operators:

- Library Color:

  - *ColorTransform*
- Library Memory:
  - *ImageBufferMultiRoi*
  - *KneeLUT*
  - *LUT*
  - *CoefficientBuffer*
  - *RamLUT*
  - *ROM*
- Library Compression:
  - *JPEG_Encoder_Gray*
- Library Transformation:
  - *FFT*

Find below information on how to adjust the settings of the affected parameters during runtime.

| **Operator Name** | **Parameters Showing Different Interface During Runtime** | **Alternative Access During Runtime** |
|---|---|---|
| *ColorTransform* | *Coefficients*:<br>• *CoefficientIndex*<br>• *CoeffiecientValue* | Field parameter *Coefficients* is replaced by separate index parameter and value parameters (parameter names: *CoefficientIndex* and *CoefficientValue*). The value is written on access to *CoefficientValue*. |
| *CoefficientBuffer* | *LoadCoefficients* | File input/output is not possible during runtime on eVA platforms. Thus, coefficient data cannot be loaded via image file (the normal use case for this parameter), but needs to be loaded as raw data.<br><br>To be able to load raw data, some additional parameters are used on eVA platforms during runtime:<br><br>*RawDataXLength*: Width of the ROI, in which the coefficients are to be defined.<br><br>*RawDataXOffset*: X position of the ROI, in which the coefficients are to be defined.<br><br>*RawDataYOffset*: Y position of the ROI, in which the coefficients are to be defined.<br><br>*LoadCoefficients*: The loading is triggered by a write cycle of value 1 to this parameter. |

| Operator Name | Parameters Showing Different Interface During Runtime | Alternative Access During Runtime |
|---|---|---|
| | | Loading into the ROI is carried out via data word *WriteRawData*.<br><br>*WriteRawData*: Data word that is written into the RAM at a write cycle to *WriteRawData*. With each write cycle, the writing position is automatically moved forward by one position within the ROI that has been defined by *RawDataLength*, *RawDataOffset* and *RawDataYOffset*. This way, a complete ROI is overwritten by continous write cycles to *WriteRawData* |
| *ImageBufferMultiRoi* | *XOffset XLength YOffset YLength RoiIndex* (additional parameter) | During runtime on eVA platforms, the parameters *XOffset*, *XLength*, *YOffset* and *YLength* are available as scalar parameter. (In the VA GUI they are defined as arrays.) An additional parameter *RoiIndex* selects the region of interest. For each *RoiIndex*, *XOffset*, *XLength*, *YOffset*, and *YLength* can be defined. |
| *KneeLUT* | *StartBasePoints*:<br><br>• *StartBasePointValue*<br><br>• *Index*<br><br>*EndBasePoints:*<br><br>• *EndBasePointValue*<br><br>• *Index* | During runtime on eVA platforms, instead of field parameter *StartBasePoints*/ *EndBasePoints*, the scalar parameter *StartBasePointValue*/ *EndBasePointValue* is available. An additional parameter *Index* selects the index of the field parameter *StartBasePoints*/ *EndBasePointValue* of the VisualApplets operator. With this common index, parameter *StartBasePointValue*/ *EndBasePointValue* sets the addressed entry in the array. For each *Index*, one *StartBasePointValue*/ *EndBasePointValue* can be defined. |
| *LUT* | *LUTcontent*:<br><br>• *Value*<br><br>• *Index* | During runtime on eVA platforms, instead of field parameter *LUTcontent*, the scalar parameter *Value* is available. An additional parameter *Index* selects the index of the field parameter *LUTcontent* of the VisualApplets operator. With this common |

| Operator Name | Parameters Showing Different Interface During Runtime | Alternative Access During Runtime |
|---|---|---|
| | | index, parameter *Value* sets the addressed entry in the array. For each *Index*, one *Value* can be defined. |
| *RamLUT* | *InitData*<br>• *InitDataValue*<br>• *InitDataIndex* | During runtime on eVA platforms, instead of field parameter *InitData*, the scalar parameter *InitDataValue* is available. An additional parameter *InitDataIndex* selects the index of the field parameter *InitData* of the VisualApplets operator. With this common index, parameter *InitDataValue* sets the addressed entry in the array. Parameter *InitDataIndex* is only available when the VisualApplets operator parameter *InitData* contains more than one entry. |
| *RamLUT* | *InitFileLoadMode InitFileName LoadInitFile* | During runtime on eVA platforms, parameters *InitFileLoadMode*, *InitFileName*, and *LoadInitFile* is not available, as the GenICam API doesn't support file input/output. |
| *ROM* | *ROMcontent*<br>• *Value*<br>• *Index* | During runtime on eVA platforms, instead of field parameter *ROMcontent*, the scalar parameter *Value* is available. An additional parameter *Index* selects the index of the field parameter *ROMcontent* of the VisualApplets operator. With this common index, parameter *Value* sets the addressed entry in the array. For each *Index*, one *Value* can be defined. |
| *JPEG_ENCODER_Gray* | *quality_in_percent* | During runtime on eVA platforms, parameter *quality_in_percent* is not available. The quantization matrix must be written directly via the parameters *quantization_matrix_index* and *Quantization_matrix_value* which are used instead of the VisualApplets field parameter *quantization_matrix* (see description of parameter *quantization_matrix* below). |
| *JPEG_ENCODER_Gray* | *quantization_matrix* | During runtime on eVA platforms, instead of field parameter *quantization_matrix*, the scalar parameter *quantization_matrix_value* |

| Operator Name | Parameters Showing Different Interface During Runtime | Alternative Access During Runtime |
|---|---|---|
| | | is available. An additional parameter *quantization_matrix_index* selects the index of the field parameter *quantization_matrix* of the VisualApplets operator. With this common index, parameter *quantization_matrix_value* sets the addressed entry in the array. For each *quantization_matrix_index*, one *quantization_matrix_value* can be defined. |
| *FFT* | | This operator currently doesn't support embedded VisualApplets devices. |

# 6. Miscellaneous

In this chapter, you find a collection of useful information for working with VisualApplets.

## 6.1. Command Line Options

VisualApplets offers some command line arguments in order to use VisualApplets, for example, in batch scripts, or for linking the program into a system environment.

The syntax follows the following scheme:

<VISUALAPPLETSINSTALLDIR>\VisualApplets <-COMMAND> <ARGUMENTS>

The commands, together with the expected arguments, are listed in the following table:

| Command Line | Argument & Meaning |
|---|---|
| -file | Execute commands stored in a specified Tcl script.<br>Arguments:<br>• Filename of Tcl script (*.tcl) |
| -compile | Load a specified file and create an applet. The design will be checked against the design rules.<br>Arguments:<br>• Filename of VisualApplets design file (*.va)<br>• Filename of resulting applet file (*.hap) |
| -merge | Merges a microEnable IV applet file with a PixelPlant applet file. A check for suitability of both source applet files will be done.<br>Arguments:<br>• Filename of mother board applet (*.hap)<br>• Filename of PixelPlant applet (*.hap)<br>• Filename of resulting applet file |
| -sdk | Generates an SDK Example (C source code) for using the applet with the Framegrabber SDK. The design will be checked against the design rules.<br>Arguments:<br>• Filename of VisualApplets design file (*.va)<br>• Filename of PixelPlant applet file (*.hap) |
| -KeepSyntheseTraceFiles | Additional option when generating a hardware applet. Keeps the intermediate files (created during the build process) for tracing purposes after the build is completed. |
| <VAFilename> | Load a specified VisualApplets design file.<br>Used, for example, when assigning *.va files for opening by double click inside the Explorer. |

| Command Line | Argument & Meaning |
|---|---|
| | Arguments:<br>• Filename of VisualApplets design file (*.va) |

Table 6.1. Command Line Options and According Arguments

## 6.2. Keyboard Shortcuts

Below, you find a list of all shortcuts available in VisualApplets.

### 6.2.1. Main Program Window

| Shortcut | Menu Item |
|---|---|
| **Ctrl+N** | **File** -> New Project |
| **Ctrl+O** | **File** -> Open Project |
| **Ctrl+S** | **File** -> Save Project |
| **Ctrl+P** | **File** -> Print Visible Diagram |
| **Ctrl+Q** | **File** -> Quit Application |
| **Ctrl+Z** | **Edit** -> Undo |
| **Ctrl+Shift+Z** | **Edit** -> Redo |
| **Ctrl+Z** | **Edit** -> Undo |
| **Ctrl+X** | **Edit** -> Cut |
| **Ctrl+C** | **Edit** -> Copy |
| **Ctrl+V** | **Edit** -> Paste |
| **Del** | **Edit** -> Delete |
| **F2** | **Edit** -> Rename Module |
| **Ctrl+I** | **Design** -> Add Operator |
| **Mouse 3+** | **Design** -> Add Operator |
| **Ctrl+F** | **Design** -> Search Module |
| **F9** | **Analysis** -> Start Simulation |
| **Ctrl+F7** | **Analysis** -> Design Rules Check Level 1 |
| **Ctrl+F8** | **Analysis** -> Design Rules Check Level 2 |
| **F7** | **Build** -> Build |
| **F5** | **Build** -> Start microDisplay |
| **F11** | **View** -> Fullscreen |
| **Ctrl++** | **View** -> Zoom In Diagram |
| **Ctrl+-** | **View** -> Zoom Out Diagram |
| **Ctrl+Mouse wheel up** | **View** -> Zoom In Diagram |
| **Ctrl+Mouse wheel down** | **View** -> Zoom Out Diagram |
| **Ctrl+Return** | **View** -> Zoom Reset |
| **Backspace** | **View** -> Window Up |
| **Ctrl+Space** | **View** -> Toggle Hand/Selection Tool |
| **Ctrl+Arrow Right** | **Window** -> Next Diagram |
| **Ctrl+Arrow Left** | **Window** -> Previous Diagram |

| Shortcut | Menu Item |
|---|---|
| **Drag+Drop** | **Project files can be dropped on main window.** |

Table 6.2. Shortcut List for Main Program Window

### 6.2.2. Simulation Viewer

| Shortcut | Menu Item |
|---|---|
| **Ctrl+Q** | **File** -> Exit Simulation Viewer |
| **Ctrl+C** | **Edit** -> Copy Image to Clipboard |
| **Del** | **Edit** -> Remove Selected Images |
| **Ctrl+Del** | **Edit** -> Remove all Images |
| **Arrow left** | **Navigation** -> Previous Image |
| **Arrow right** | **Navigation** -> Next Image |
| **Ctrl++** | **View** -> Zoom In |
| **Ctrl+Mouse wheel up** | **View** -> Zoom In |
| **Ctrl+-** | **View** -> Zoom Out |
| **Ctrl+Mouse wheel down** | **View** -> Zoom Out |
| **Ctrl+Mouse 3** | **View** -> Normal Size 1:1 |
| **F11** | **View** -> Full Screen |
| **Ctrl+F** | **View** -> Fit to Window |
| **Ctrl+T** | **View** -> Mapped Thumbs |
| **Ctrl+R** | **View** -> Reset Parameters |
| **Drag+Drop** | **Images can be dropped directly on simulation viewer (source) or simulation source module.** |

Table 6.3. Shortcut List for Simulation Viewer

## 6.3. Error Reporting

Even the best developed and tested software might – at times – run into some errors. If you want to help Basler in improving VisualApplets, please report any misbehavior to our support team and describe the situation.

Additional data like the VisualApplets design file (*.va), a description of the system environment like operating system and PC hardware, or your specific VisualApplets system settings is helpful for us when correcting the erroneous misbehavior.

In case of system crashes, a trace file will be generated. This file allows us at Basler to restore the exact situation that led to the crash. This trace file is stored inside a zip-archive which you will find in the installation directory of VisualApplets: <INSTDIR>\bin\bugs\bug*.zip. Please decide if you want to send this supplementary information to Basler or not. Please note that the trace file contains only information that has been entered into VisualApplets. It does not contain any information regarding the computer system VisualApplets was running on.

# Part II

Tutorial and Examples

# 7. Introduction

VisualApplets is a very easy to learn self explaining software. The usage is very simple. However, as for all programs and programming languages, examples and step by step tutorials help learning its usage. We recommend to read and implement the design presented previously in the getting started chapter to get introduced into the world of VisualApplets (see 2. *Getting Started*). Next, looking at the tutorials and examples presented in the following will ease the barrier to start your own implementations.

Always mind to use the operator reference if you need more information about VisualApplets operators (see Part III, 'Operator Reference'). Each operator description includes a list of examples where it is used.

If you have problems of understanding a feature of VisualApplets or if you have problems in understanding the theory behind VisualApplets desings, check the respective chapters in the user manual (see Part I, 'User Manual').

The tutorial and examples consist of three parts. First, a tutorial presenting the basic design theory is presented. Next, a list of basic acquisition applets is given which will allow you to start your own project with the required camera interface and hardware device. The examples chapter is a list of example files which can be found in the VisualApplets installation folder. These examples show the usage of operators and might provide implementation ideas.

# 8. Hardware Applet: From Idea to Application

In this Application Note you will learn how to design and create an image processing application (=applet) in VisualApplets and how to use it on a frame grabber. You will also learn how to use this applet during runtime using microDisplay and Framegrabber API and how to configure parameters of your applet in this software. You find an introduction how to design, build and use an applet during runtime in microDisplay at 2. *Getting Started*. In this Application Note you will learn the above mentioned phases of applet development and usage with the example of a Sobel edge filter application.

## 8.1. Workflow Description

The workflow is as follows:

1. Designing an applet in VisualApplets.
2. Simulating the applet in VisualApplets and performing a Design Rules Check (DRC).
3. Building the applet.
4. For mE5 platforms: Flashing the applet.
5. Using the applet on the hardware via microDisplay or Framegrabber API.

## 8.2. Designing an Applet in VisualApplets

### 8.2.1. Starting a New Project

The design phase of an applet takes place in VisualApplets. To open VisualApplets, click on the file `VisualApplets.exe` in the `bin` folder in the VisualApplets installation directory, or the VisualApplets program icon in the Windows Start menu or on the icon on your desktop.

1. At first, the VisualApplets main window opens up with no project loaded.

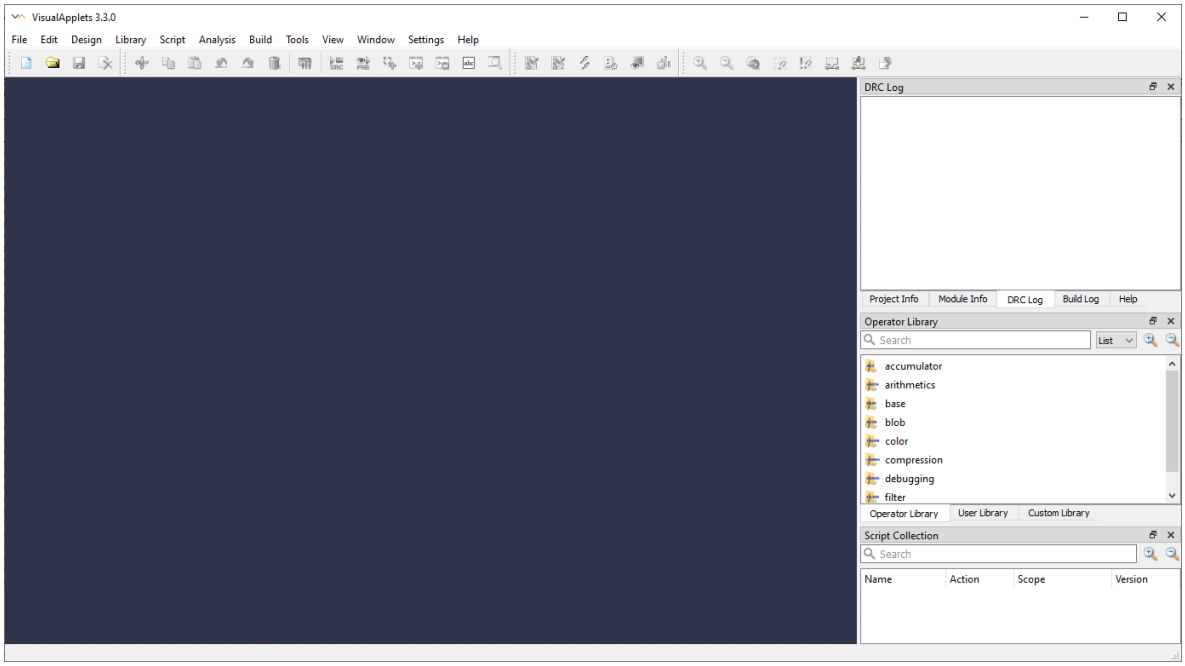



Figure 8.1. VisualApplets Main Window

2. Click on **File**+**New ( Ctrl+N )** or use the **New** icon from the **File** icon bar. A **New Project** window opens up which allows you to specify project name, target hardware platform, and target runtime. You can always change these settings later on.

3. To follow the example here, use the following settings:

   **Project Name: Sobel_Filter**

   **Hardware Platform: microEnable 5 marathon VCL**

   **Target Runtime: Win 64 (Windows/AMD64)**

4. Confirm your settings by clicking **OK**.

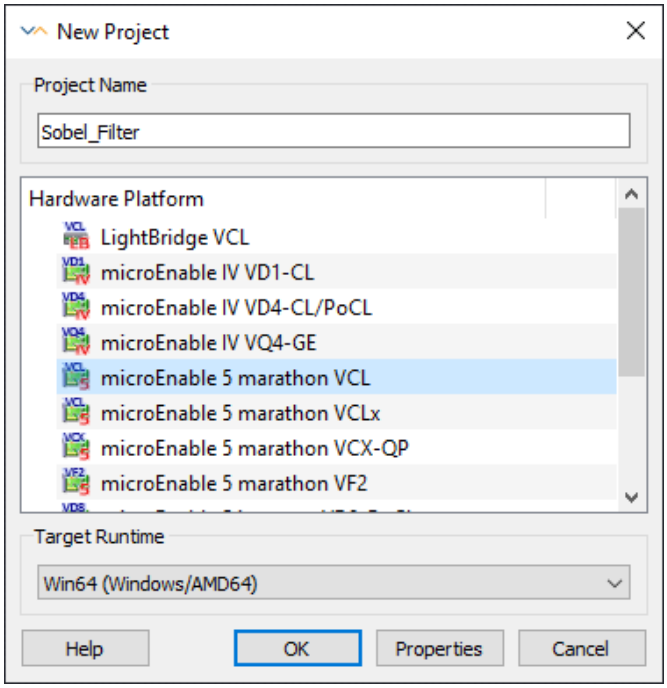



Figure 8.2. New Project window

VisualApplets now starts a new project and you see a blank design window in the center of the program window. In the **Project Info** tab on the right, information regarding the current project, such as project name, target hardware, target platform etc. is displayed.

### 8.2.2. Operators and Links

In VisualApplets, image processing operations are represented by operators. You find all these operators in the **Operator Library** on the right side of the VisualApplets design window. You can very easily place operators into the design window using drag-and-drop.

An instance of an operator in the design is called a **module**. Operators can have input and output ports. Operators in a design (i.e., modules) can be connected using these ports. Connections between modules are called links which are represented in the design window by arrows. These modules and links represent the image- (or signal-) processing pipeline. Hence, the order of operations is determined by the order of modules.

The operators and links have properties, which describe the settings relevant for the current image processing pipeline, like image dimensions. To see and edit these properties, double-click the operators. The properties of the operators are explained in detail in the operator documentation under **Help**.

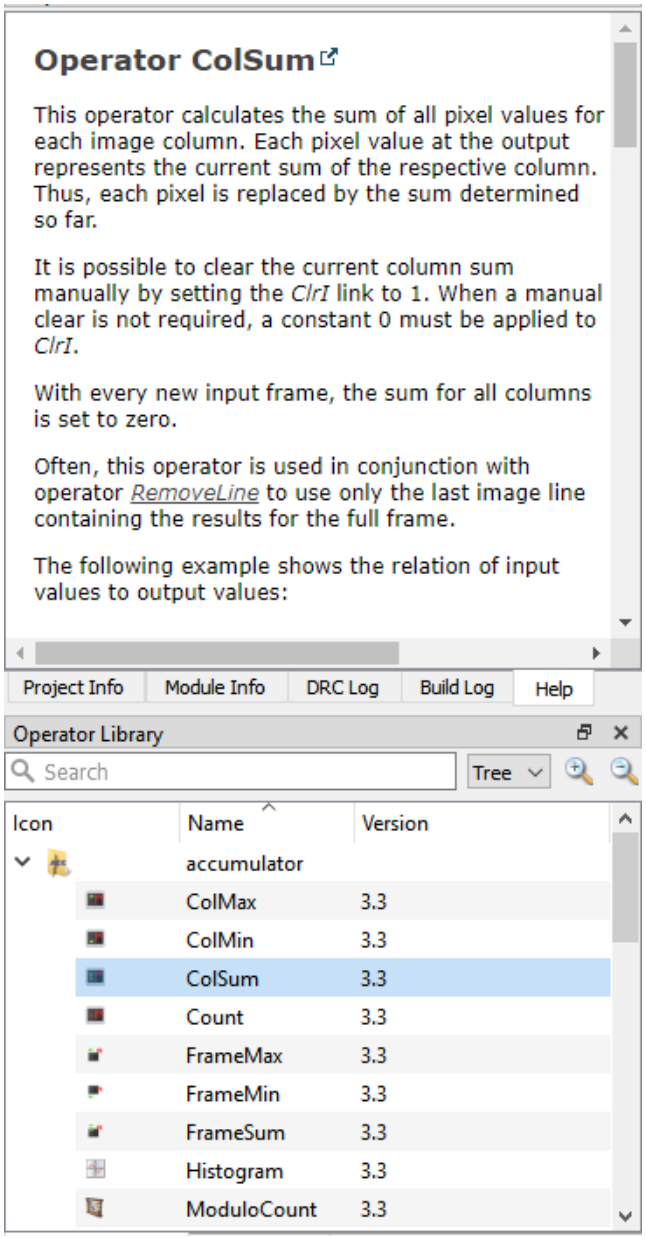



Figure 8.3. Operator Documentation in VisualApplets

### 8.2.3. Implementation of the Design

This example implements a Sobel edge filter algorithm [https://en.wikipedia.org/wiki/Sobel_operator].

In the VisualApplets design sample library, there are "ready-to-use" samples for edge filtering and further image processing applications. You find these design examples under `Examples\Processing` in your VisualApplets installation directory with the corresponding documentation, see 12. *Processing Examples*. You can use these example designs as base for your own application.

For the implementation of this sample Sobel filter design, locate the operators in the **Operator Library**, drag them into the design and connect them with links as described in the 2. *Getting Started* section.

You can rename the modules later on for better overview in the design and for better usage during runtime.

Whenever useful, structure the design using HierarchicalBoxes for a better design overview.

And don't forget to save the design from time to time.

## 8.2.4. Design Components

The sample design in this application note consists of the following components as shown in the following figure:

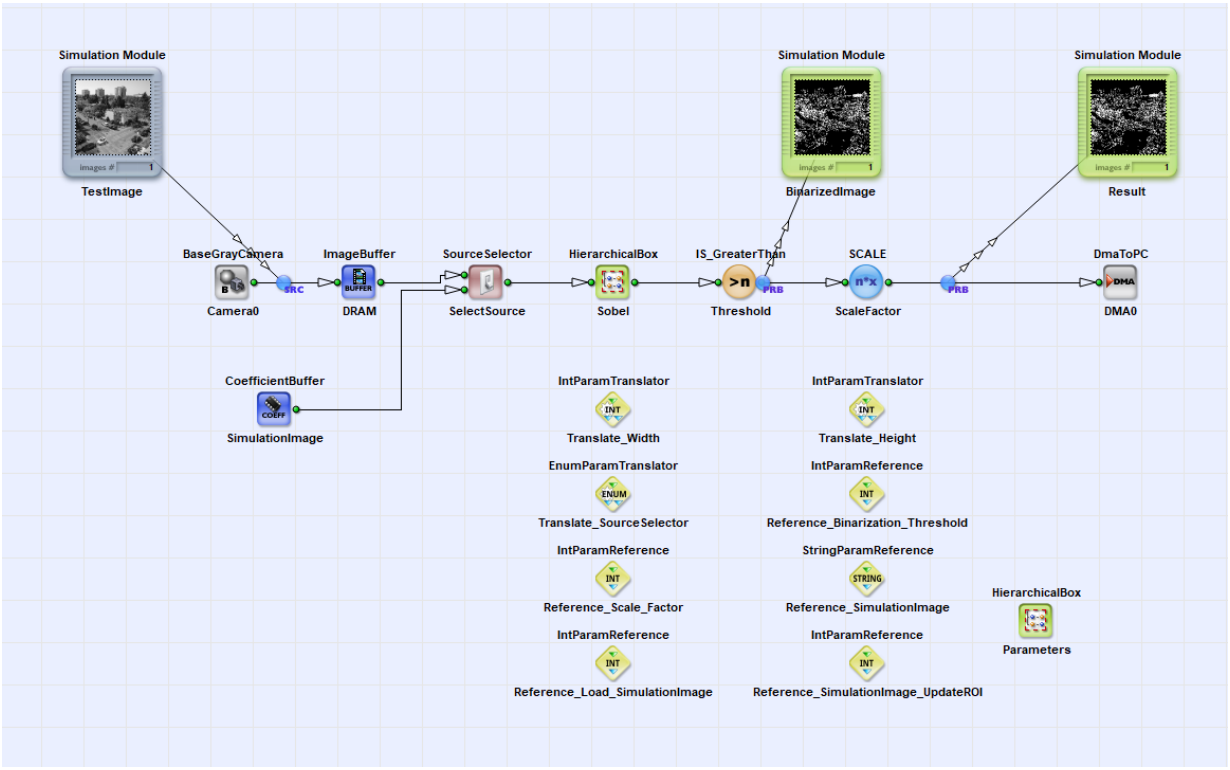



Figure 8.4. Example Design Implementation `Sobel_Filter.va`

- A Camera Link grayscale camera interface
- An image buffer
- The Sobel edge filter module with edge filtering in x and y direction (based on the example `Sobel_Multi_Gradient.va` under `Examples\Processing\Filter\EdgeDetection\Sobel_Multi_Gradient` in the VisualApplets installation directory)
- An adjustable binarization threshold and a scale operator for better visualization of the binarized image on the display during runtime
- The DMA to PC.

You can use the *CoefficientBuffer* in the design as simulation source for images instead of using the images acquired by a camera. Via the operator *SourceSelector* you can choose the image input source for the processing pipeline in VisualApplets and during runtime.

## 8.2.5. Parameter Settings

To parameterize the operators and links, you can edit the parameters of each element by double-clicking the element.

### 8.2.5.1. Link Parameters

Links have properties, which define the properties of the processed images such as maximum image dimensions, the image protocol, the bit width, and a parameter which is related to the maximum

possible bandwidth: the parallelism (i.e. numbers of pixel transferred at one design clock cycle). If these links are connected to specific operators (see corresponding operator documentation), e.g. the camera interface operators, you can edit and adjust these properties according to your purpose.

The higher the maximum image dimensions and the parallelism you choose, the higher the FPGA resource consumption is. Therefore, Basler recommends to set these values only as high as necessary in order to save FPGA resources. The link properties can only be changed during design phase in VisualApplets.

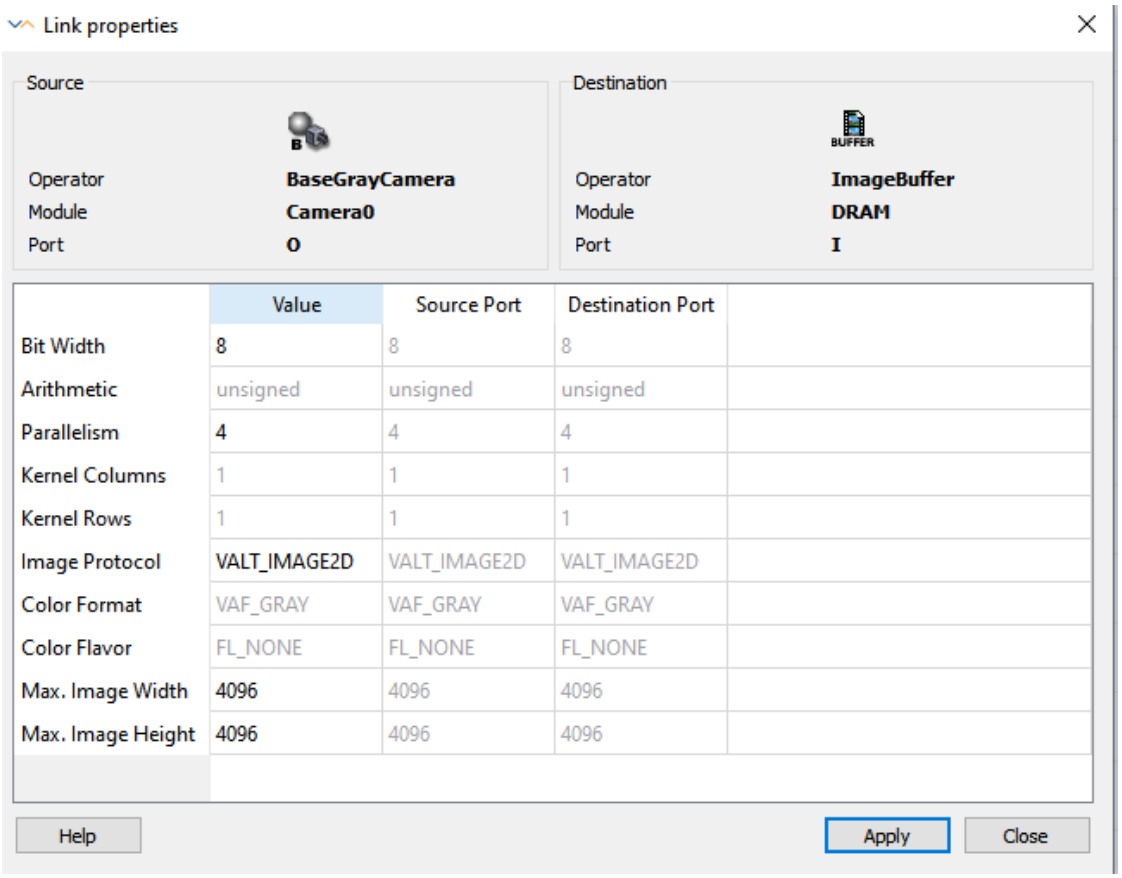



Figure 8.5. Link Properties

## 8.2.5.2. Operator Properties

There are two types of operator properties: **static** and **dynamic**. You can change static parameters only during design phase in VisualApplets, whereas you can change dynamic parameters during design phase and during runtime. See the screenshot below for examples of static and dynamic parameters for the operator *IS_GreaterThan*.

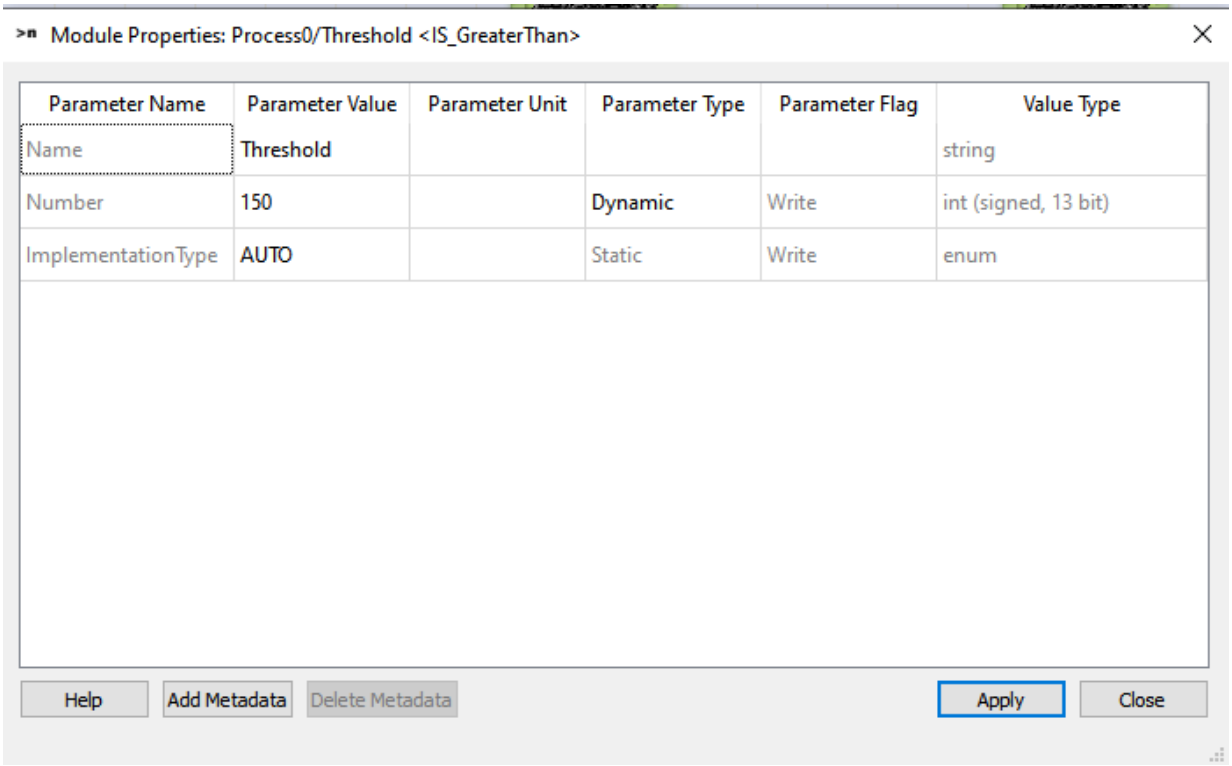



Figure 8.6. Static and Dynamic Operator Parameters

In this sample design, set the following parameters:

Operator *IS_GreaterThan*: Set the parameter *Number* to *600*.

This identifies the grayscale of the edges. If you want to find out the grayscale of your edges, set a simulation probe before the *IS_GreaterThan* operator, run a simulation, and zoom into the pixels that contain edges and see which grayscale they have. This is a dynamical paramter, which you can alter during runtime.

Operator *SCALE*: Set the parameter *ScaleFactor* to *255*.

This augments the visibility of the edges. This is a dynamical parameter. However, to save FPGA resources, you can set this parameter to *Static*.

### 8.2.5.3. Preparation for Parameter Access during Runtime

During runtime you can change the dynamic parameters. A common use case is that you have many operators in a design and you need to change the same parameters e.g. the image dimensions for many different operators in the design.

The library **Parameters** provides operators which enable accessing different module parameters in a design by controlling only one parameter. Additionally, these operators also enable that access parameters are displayed on a specific hierarchy level. These options make parameter configuration easier during runtime. Whether the library **Parameters** is available for you depends on the license you have purchased. For questions regarding your license, contact the Basler Sales Department [https://www.baslerweb.com/en/sales-support/sales/]. The example design `Sobel_Filter.va` (see Figure 8.4, 'Example Design Implementation `Sobel_Filter.va`') uses translate and reference parameters to easily access image width and height, the image source selector, the threshold value and the scale factor on the hierarchy level **Parameters**. For this, create an empty HierarchicalBox, name it **Parameters** and set the *DisplayHierarchy* parameter of the corresponding translate and reference parameters to **Parameters**. You find a detailed documentation of the **Parameters** library in the corresponding documentation, see 28. *Library Parameters* [1049].

## 8.2.6. Finalizing the Design

### 8.2.6.1. Simulation Sources and Probes

During design phase the simulation sources and probes are very helpful. Using the simulation sources you can load test images and check your processed image on every link in the processing pipeline within a few moments. Exception: Signals can’t be simulated. Furthermore, the simulation performs a Design Rule Check Level 1 for the formal correctness of the implementation. The simulation result is equivalent to the result during runtime. You find the simulation sources and probes in the icon menu or under Analysis in the text menu in VisualApplets. For a detailed description of the usage of the simulation sources and probes, see the detailed documentation under Section 3.10, 'Simulation'.

In Figure 8.4, 'Example Design Implementation `Sobel_Filter.va`' you can see a test image loaded to a simulation source in the example design `Sobel_Filter.va`. After threshold and before DMA you can see the result of the current processing step.

### 8.2.6.2. Design Rule Check (DRC)

After you have finalized the implementation, it is recommended to perform a Design Rule Check Level 1 and 2. To perform a Design Rule Check Level 1 and 2, select **Analysis**+**Design Rules Check Level 1 and 2** or use the icon **Design Rules Check Level 1 and 2** from the icon bar.

Design Rule Check Level 1 reports formal errors in the implementation and their exact location in the design. You must correct these errors before the design can be translated to a hardware applet.

Design Rule Check Level 2 additionally gives information about the estimated FPGA resources used in the design. With this information you can check, whether enough FPGA resources are available for your image processing implementation. If the resources exceed 100% of one of the FPGA resources, you need to redesign your implementation to save resources.

In Figure 8.7, 'Design Rule Check 1 and 2 for the Example Design `Sobel_Filter.va`' the result of Design Rule Check 1 and 2 is shown: Everything is designed correctly and about 36% of the available Lookup Tables, 22% FlipFlops, 15 % Block RAM and 2 % of the available embedded ALUs are used. You can see a detailed overview on how many FPGA resources each element in the design consumes (see Figure 8.8, 'FPGA Resource Estimation') under **Analysis**+**FPGA Resource Estimation**

DRC Log

**0 Errors**
**0 Warnings**

10:00:23 -- **Design Rules Check Level 1 successfully performed.**

## Check Build Preconditions

**Creation date:** Monday, 11 April 2022 10:00:23

10:00:23 -- **All checks successfully performed.**

## Netlist Generation

**Creation date:** Monday, 11 April 2022 10:00:23

Netlist generation started...

Info: Building design for FPGA clock frequency 125 MHz

**FPGA resource estimation:**

| Resource | Count | Fill Level |
|---|---|---|
| LookupTables | 37132 | ~ 36% |
| Flip Flops | 44960 | ~ 22% |
| Block RAM | 103 | ~ 15% |
| Embedded ALU | 14 | ~ 2% |

10:00:26 -- **Netlist successfully generated in 00:00:03.**

Figure 8.7. Design Rule Check 1 and 2 for the Example Design `Sobel_Filter.va`

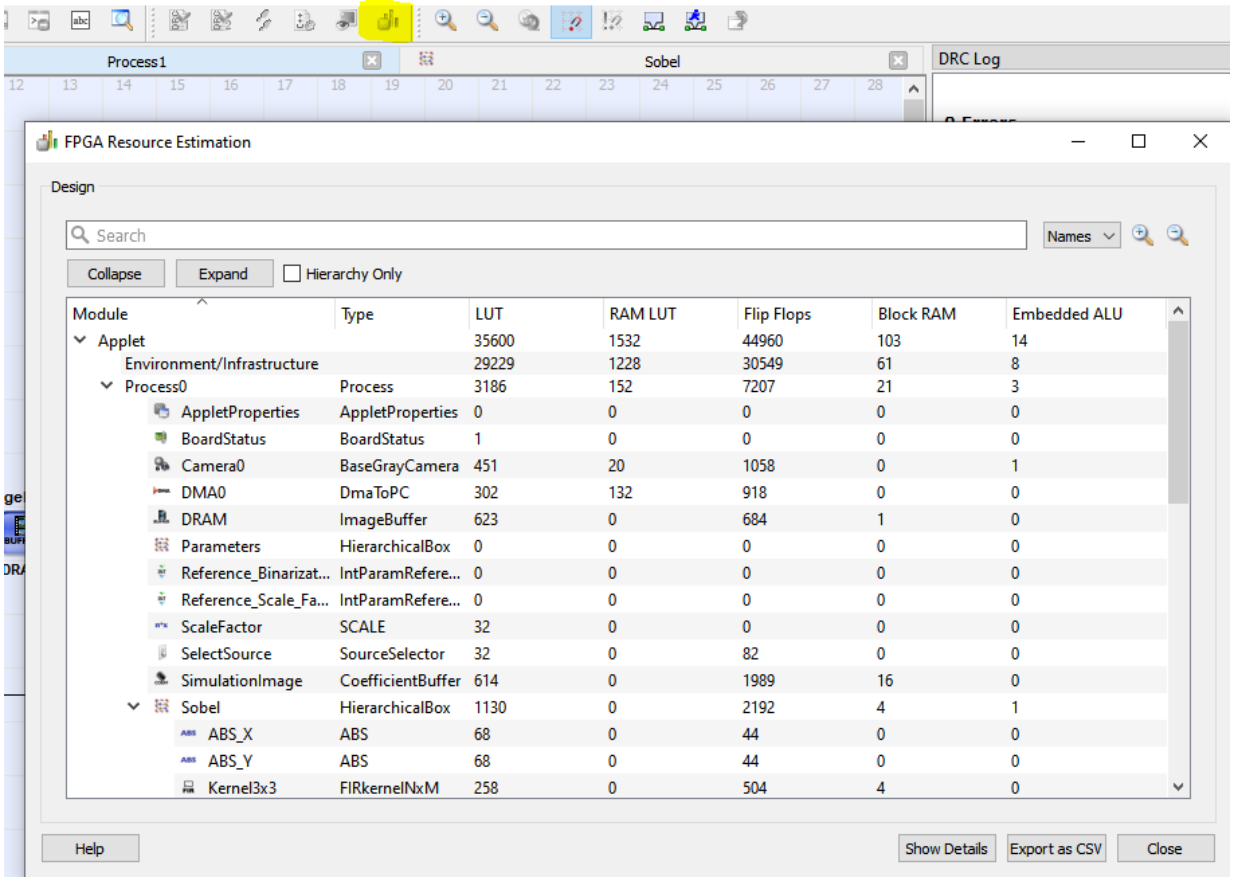



Figure 8.8. FPGA Resource Estimation

## 8.3. Building the Applet in VisualApplets

### 8.3.1. Precondition

The hardware applet build process can only be performed if the XILINX tools are properly installed. Under https://docs.baslerweb.com/visualapplets/installing-visualapplets#which-xilinx-toolchain-and-version-for-which-frame-grabber-platform you can find a detailed overview, which Xilinx versions is suitable for your frame grabber platform. In this sample design `Sobel_Filter.va` the microEnable 5 marathon VCL platform is selected. Thus, Xilinx ISE version 14.7 or Vivado versions between 2016.1 and 2020.1 are suitable.

### 8.3.2. Editing the Build Settings

Now you have finished the example design `Sobel_Filter.va` and want to translate the design into a hardware applet. For this translation process, called **build** you need to select the correct Xilinx build settings. Open **Settings**+**Build Settings**, and select the batch file `settings64.bat` of your corresponding Xilinx version (recommended: Xilinx Vivado 2018.2) from your file system and confirm with OK. You find a detailed description of editing the build settings under Section 4.9, 'Build Settings'.

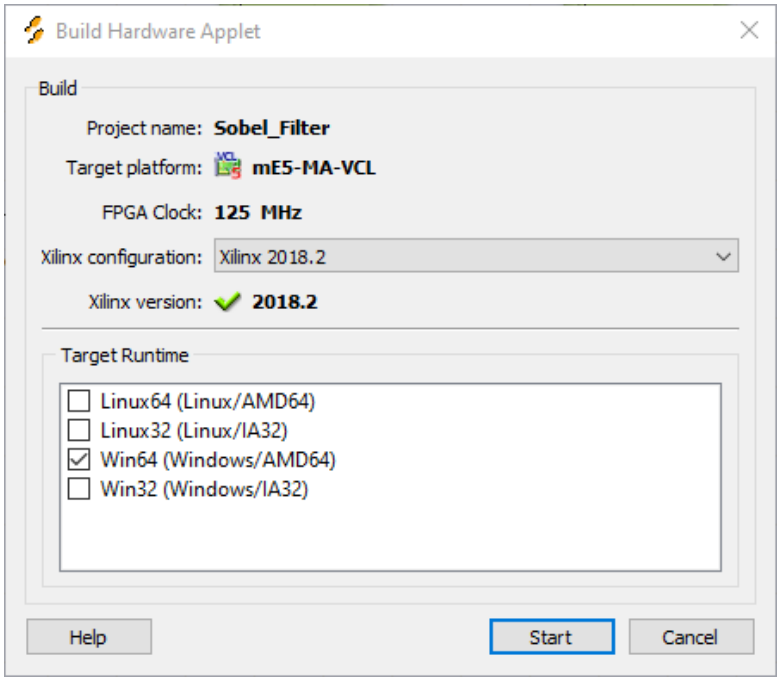



Figure 8.9. Build Hardware Applet Dialog

VisualApplets now uses the Xilinx tools to translate the application into the FPGA bitstream, i.e., the **program** or **applet**. The duration of this process depends on the complexity of the design. The build of highly complex designs might take several hours. For this example implementation `Sobel_Filter.va`, the build time is about 15 minutes. After successful build, the applet is fully generated. The name of the applet (`*.hap` file) is the same as the name of the design file (`*.va`): `Sobel_Filter.hap`

## 8.4. Running the Applet on Hardware

### 8.4.1. Precondition

To run an applet in hardware on a frame grabber, a programmable (V Series) frame grabber (hardware) needs to be installed on the system. Furthermore, the Framegrabber SDK software needs to be installed on the PC. For this example applet for the microEnable 5 marathon VCL, the latest possible Framegrabber SDK version is 5.7.

### 8.4.2. Flashing

If, like in this example, you use a frame grabber of the microEnable 5 series (marathon, ironman or a LightBridge), you need to flash your frame grabber with the new applet using microDiagnostics.

If you are going to use the new applet on a microEnable IV frame grabber or a frame grabber of the latest CXP 12 series, just skip this section.

To flash the frame grabber, perform the following steps:

1. Start the tool microDiagnostics under `bin` in the Framegrabber SDK installation directory.
2. Select your frame grabber.
3. Select **Firmware** in the left side of the dialog.
4. Assign your hardware applet to one of the displayed partitions (0 to 7): Go to the directory, where the created hardware applet is located and select the file.

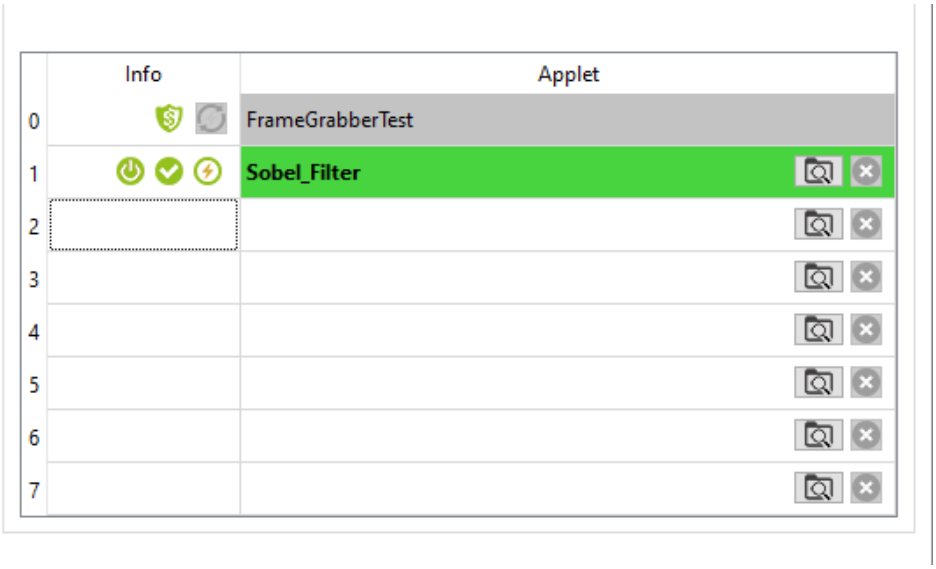



Figure 8.10. Firmware Partitions Displayed in microDiagnostics

5. Click **Flash Now**

6. Wait until the new firmware is completely installed. When flashing is completed, you get a message in microDiagnostics.

7. Follow the instructions in the message.

### 8.4.3. Testing and Loading the Applet in microDisplay

To test your applet `Sobel_Filter.hap` and to set some first parameters,use the program microDisplay. Following steps are necessary to load an applet:

1. Save the applet (here: `Sobel_Filter.hap`) under `Hardware Applets\<your frame grabber>` (here: mE5-MA-VCL) in the Framegrabber SDK installation directory.

2. Start microDisplay from the `bin` folder in the Framegrabber SDK installation directory.

3. In microDisplay, select the frame grabber you want to use under **Acquisition Devices**. Immediately, all applets available for the selected frame grabber are displayed.

4. Select the `*.hap` file you want to use (here: `Sobel_Filter.hap`) with double-click.

### 8.4.4. Parameter Settings and Acquisition

After loading the applet, you can see the parameter tree and the image acquisition window in microDisplay. Below, a screenshot of the applet `Sobel_Filter.hap` loaded in microDisplay is shown.

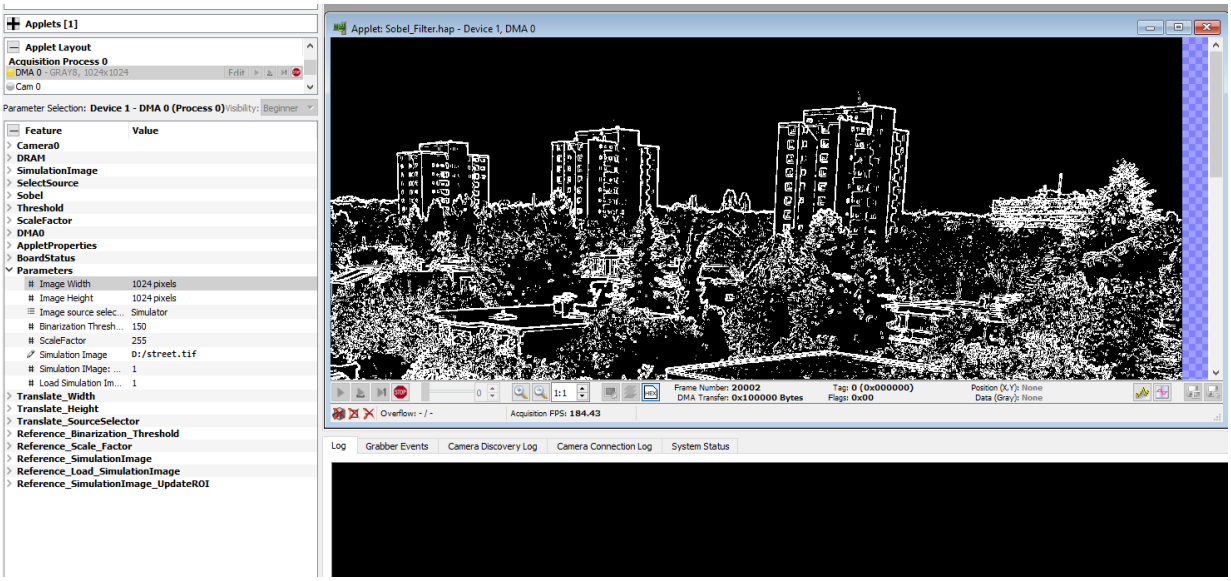



Figure 8.11. Parameter Tree and Image Acquisition Window in microDisplay

On the left side, the parameter tree is displayed. These parameters are equivalent to the operator names during design phase in VisualApplets. It is important to use a good naming of the operators, in order to find the correct parameters you want to adjust for acquisition. Dynamic parameters are adjustable during runtime. There are two kinds of dynamic parameters: Those parameters, which are only adjustable before start of acquisition (e.g. *DMA dimensions*) and those, which can be adjusted even during image acquisition (e.g. threshold values of operator *IS_GreaterThan*).

In this example, you perform the image acquisition using an image from a simulator source and set the binarization threshold to value 150. If you have integrated the parameters of the **Parameters** library during design phase in VisualApplets, you can set all relevant dynamic parameters under the hierarchy level **Parameters** in the parameter tree. Otherwise, you need to step though the different hierarchy levels to find the correct parameters we want to set.

In this example, go to the hierarchy level **Parameters** and unfold it:

1. Right mouse-click the **Parameter Simulation Image**. As a result, you can select the example image you want to use for image acquisition.

2. Load this image by clicking **Load Simulation Image**.

3. Set the parameter *ImageSource selector* to *Simulator* and the *Binarization Threshold* to value *150*.

4. Set the image dimensions to 1024x1024 pixels.

After you have set these parameters (or all parameters relevant for your applet), you can start the acquisition. Here you can select between continuous grabbing, grabbing of a sequence or a single frame. To stop the continuous grabbing, click on the Stop button. For the case of a multi process design, you can start and stop the the acquisition of the single processes separately.

For a detailed description of the applet configuration and acquisition with microDisplay, see https://docs.baslerweb.com/frame-grabbers/what-is-micro-display-x.

## 8.4.5. Starting the Applet in Your Own SDK

Instead of performing the applet configuration and acquisition in mircoDisplay X you can alternatively integrate them in your own SDK environment. In this section you find an introduction to the basic functionalities of the relevant SDK componets for image acquisition and configuration.

As base of a C++ SDK you can use the VisualApplets internal SDK code generator, which you find the icon menu or under **Build**+**Generate SDK Example** in VisualApplets. This function automatically

generates the corresponding CMake File and the VisualStudio project files, which you can use as base for your SDK code.

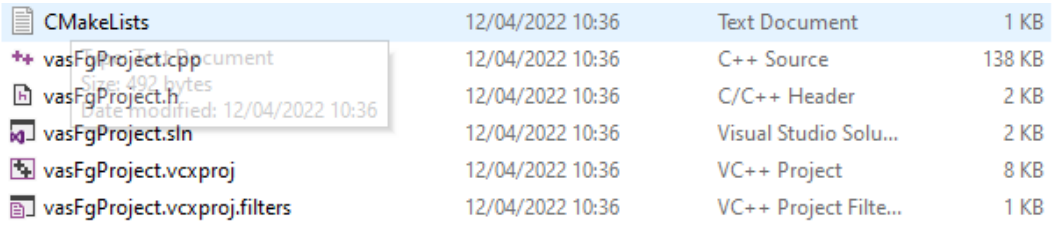



Figure 8.12. Generated SDK Project Files

The generated C++ code in the project file `vasFgProject.cpp` contains the following code components:

1. Frame Grabber: Initialization

   Definition of number of acquisition cycles, definition of board index, call of function `Fg_Init("Sobel_Filter.hap", boardIndex)` to start the applet.

2. Allocation of memory for buffers for acquisition.

   function: `Fg_AllocMemEx(fg, ..., ...)`

3. Frame Grabber: Get* and Set* parameters for Process0

   functions: `Fg_setParameterWithType(fg, ..., ..., ...)` and `Fg_getParameterWithType(fg, ..., ..., ...)`

   Here all parameters contained in the design are listed. Basler recommends to delete all parameters, which are not necessary for adjustment during runtime, for better overview.

4. Create the display(s)

   function `CreateDisplay()`

5. Start acquisition at applet and camera for each present port:

   function: `Fg_AcquireEx(fg, 0, nrOfCycles, ...,...);`

6. Grab images:

   function: `Fg_getLastPicNumberBlockingEx(fg, ..., ..., ..., ...);`

7. Stop acquisition

   Function `Fg_stopAcquireEx(fg, .., .., ...)`

8. Close the display

   function `CloseDisplay(...);`

9. Release the memory for buffer(s)

   function `Fg_FreeMemEx(fg, ..);`

10. Frame Grabber Uninitialization

    function `Fg_FreeGrabber(fg);`

A detailed description and explanation of the SDK functions is available at https://docs.baslerweb.com/frame-grabbers/sdk/basler__fg_8h.html

An introduction to the Framegrabber API is available at https://docs.baslerweb.com/frame-grabbers/framegrabber-api.html.

You can integrate the generated sample SDK code into a larger SDK environment for the further software processing of the frame grabber (pre-)processed images.

# 9. Basic Design Theory

The following tutorial will guide you through the basic principles of VisualApplets step by step. Don't miss the getting started tutorial before you continue with the following sections. The getting started tutorial can be found in 2. *Getting Started*.

1. Section 9.1, 'Applet Parameterization'

   The first tutorial uses the VisualApplets design implemented in the getting started guide. It will show the parameterization of operators and links in a more detailed context. You will learn how settings can influence the design and you will learn on how to solve conflicts.

2. Section 9.2, ' Multiple DMA Channel Designs '

   In this tutorial, the use of multiple DMAs in one design is shown. In detail, a threshold binarization as well as an image selection is implemented. You will learn how to binarize images and how to count and remove images from a sequence.

   The example will include all required steps of the development including implementation, verification by using the simulation and the usage in hardware.

3. Section 9.3, ' Synchronization of Asynchronous Image Pipelines '

   The third tutorial outlines two synchronization examples. You will learn how to merge the images of two camera sources. One example will present an image overlay of the images of port B to the images of port A. The second example shows a simple image stitching.

## 9.1. Applet Parameterization

Let's have a look on the design of the getting started tutorial once again. If you have not made the design of the getting started so far, you should go back and do the getting started first (2. *Getting Started*). So far, the only adaptation we made were the parameter names. Now, we want to adapt the applet to process images from cameras with the following properties

- image side size 512 x 512
- 12 bit / pixel grayscale
- Camera Link interface in dual tap base configuration mode

Our design is already equipped with a CameraGrayAreaBase operator. This operator is from the mE4VD4-CL operator library and is only available for mE4 Camera Link frame grabbers, i.e., the mE4VD4-CL library is a library containing hardware specific operators. The CameraGrayAreaBase operator supports grayscale area scan cameras using Camera Link in base configuration mode. Double click on the camera module in your design to open it's properties. One of the parameters is called "Format". Here, the specific Camera Link format can be selected. We select DualTap12Bit, apply our changes and close the window. Our design is now capable to acquire 12 bit camera images.

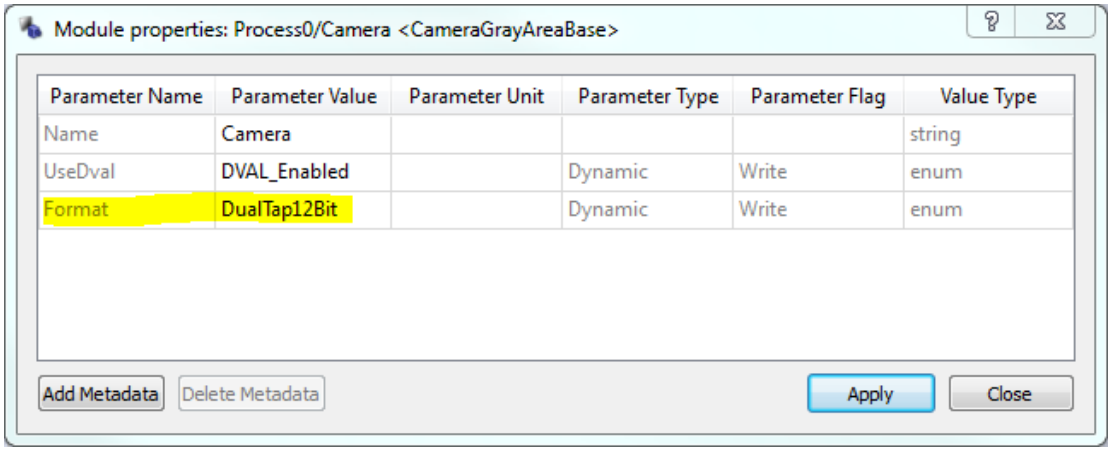



Figure 9.1. Properties of Operator CameraGrayAreaBase

Note that the "Format" parameter is dynamic. This means, the value can be changed after synthesis of your design in hardware which allows the flexible use of multiple camera formats. However, if we double click on the link at the camera, we see that the link's bit width is still at 8 bit/pixel. Moreover, the maximum image dimension is not large enough. You might ask yourself why a 12 bit camera can be used if the link only has 8 bits. The answer is that only to eight most significant bits of the camera images are transported i.e. only the available link bits are used. The operator will automatically select the correct bits from the images depending on the settings in the camera operator. Check the documentation of the camera operators for more information (Section 29.12, 'CameraGrayAreaBase'). We will change the bit width to 12 bit to process the full camera bit depth.

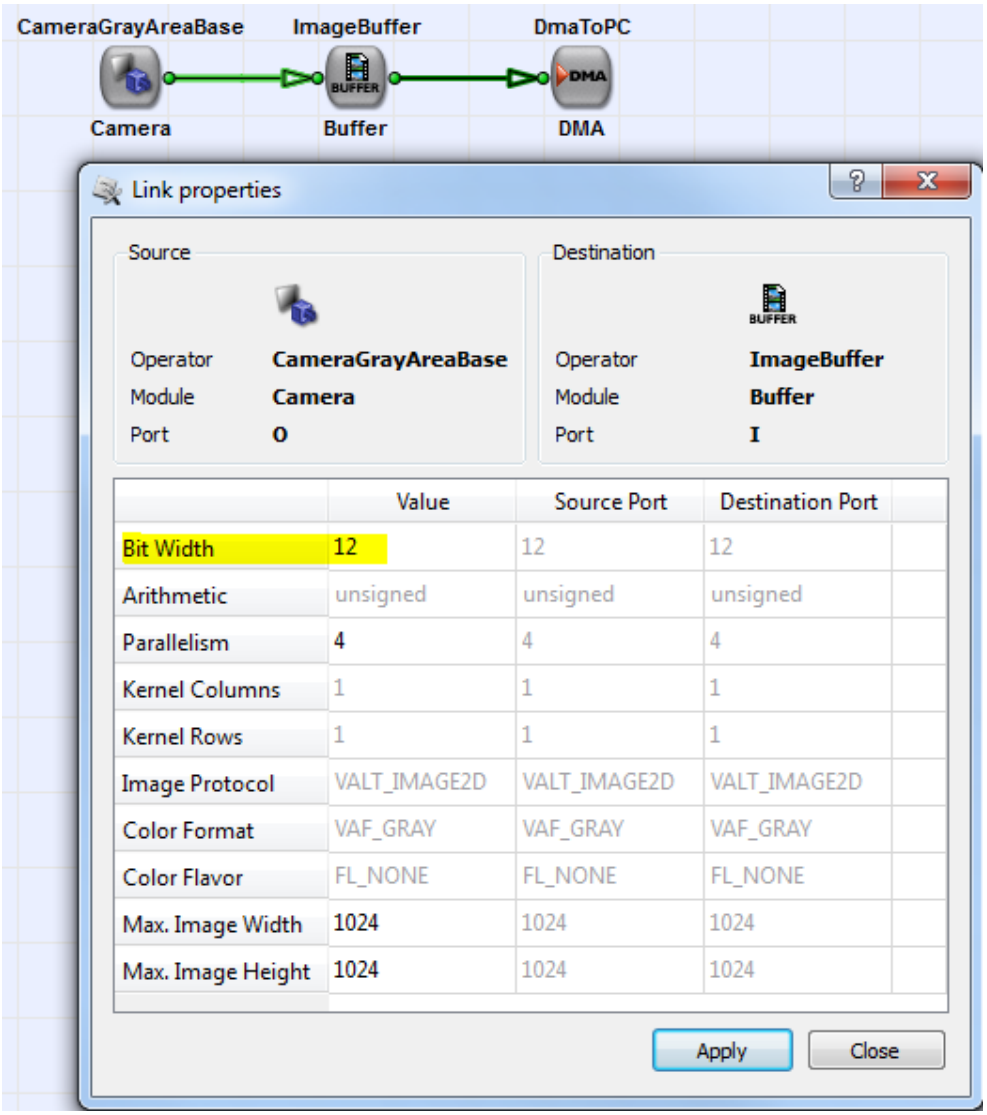



Figure 9.2. Changed the Link Bit Width of the Camera Operator

After changing the value, click on Apply. As you can see, the color of the links between the camera, the buffer and the DMA has changed from black to green. This indicates that the change was accepted and validated. Moreover the link property "bit width" was propagated through the image buffer to the link between the buffer and the DMA module. Thus, users are not required to change the bitwidth on every link. Link properties are propagated through modules until they change the link bitwidth themselves or require user input. If you open the link properties of the link between the buffer and the DMA, you will notice that it is not possible to change the bitwidth here.

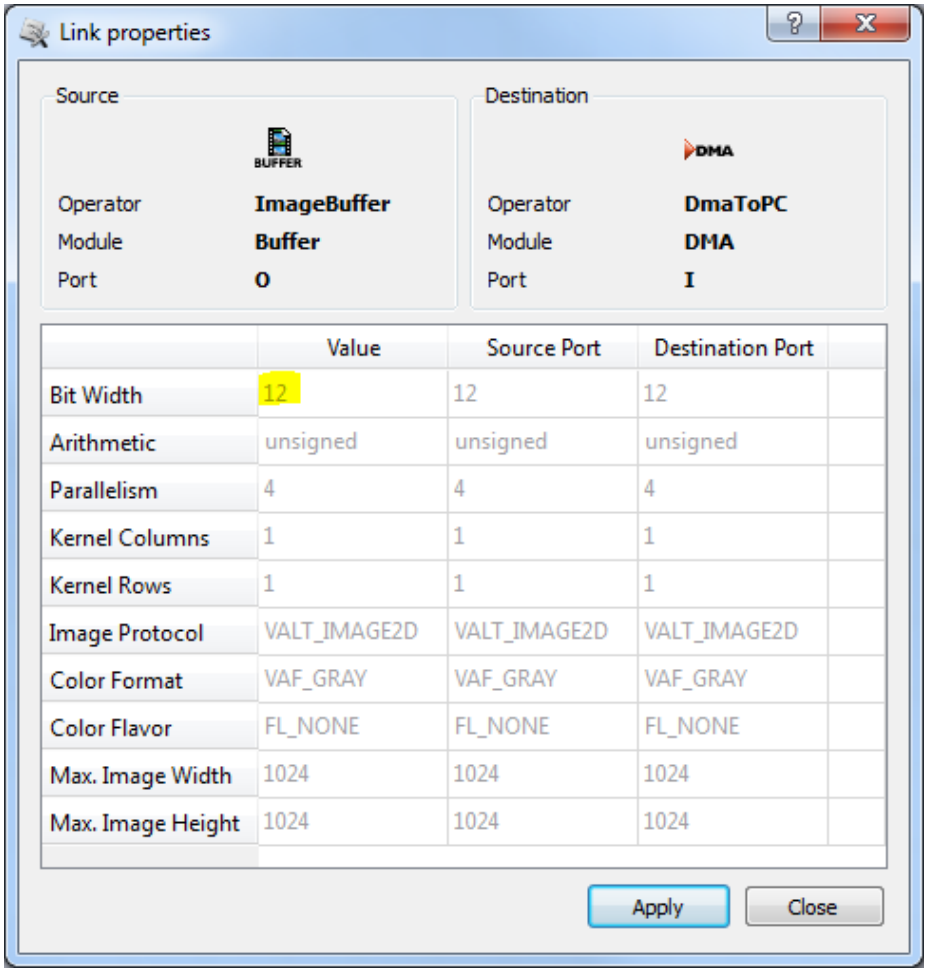



Figure 9.3. Bit Width Cannot be Changed at Buffer Module Output Link

That's because the antecedent operator *ImageBuffer* is not capable to change the link bit width. Hence, link properties can only be changed if the connected operator allows the change i.e. is by definition an operation which influences the bit width.

The required image side size for the camera is 512 x 512 pixel. In the link properties, we can see properties called Max. Img Width and Max. Image Height. These properties represent the maximum image dimension on the link. This is the maximum. Any lower dimensions are allowed. However, an image size exceeding this limitation will violate the VisualApplets design rules and you might get an error when using the applet in hardware.

The setting of the image dimension should always be chosen to the required minimum as a lower image dimension can save FPGA logic and memory resources. Back to our example: Open the link properties of the link between the camera and the buffer and select a maximum image width and height of 512 x 512. After you are done, click on apply.

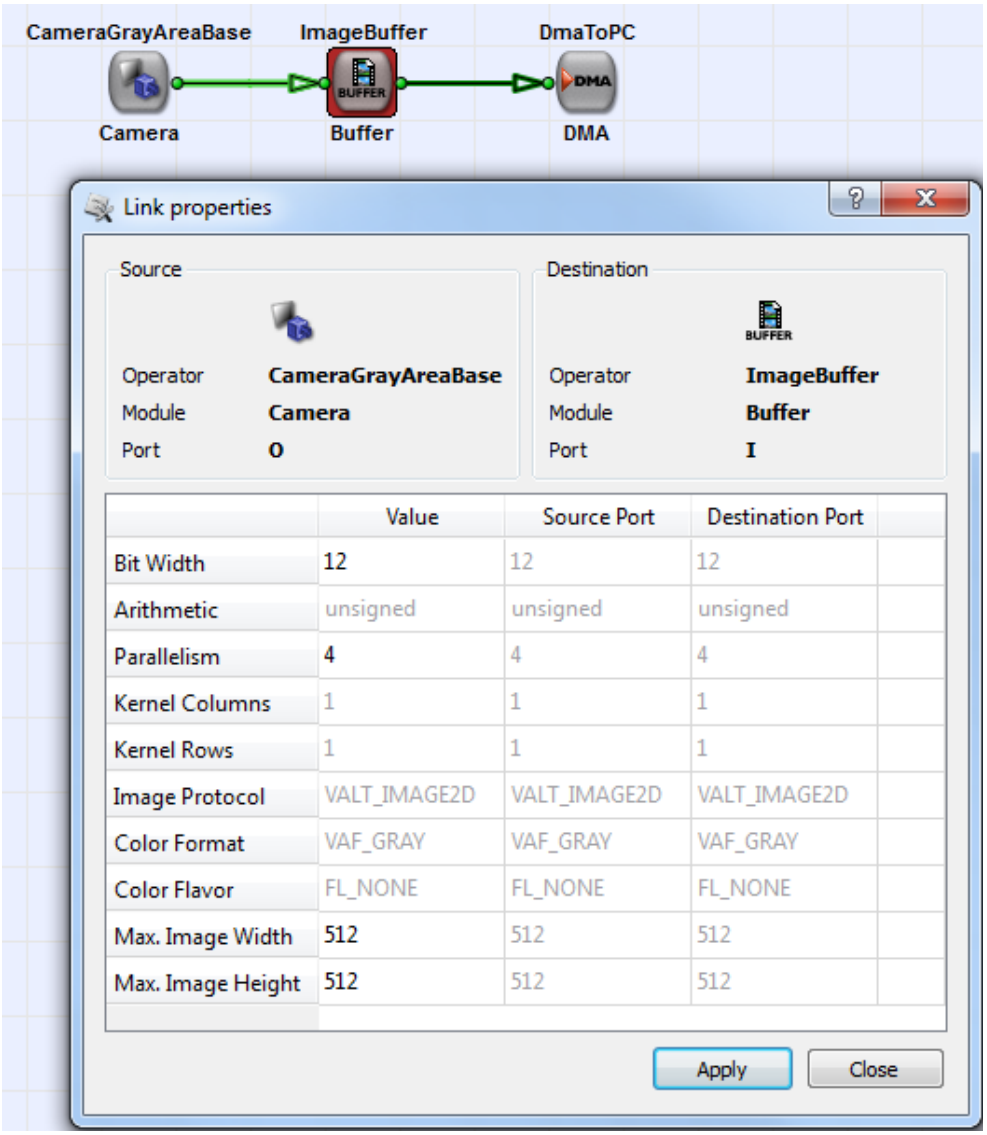



Figure 9.4. Illegal Condition after Link Property Change

Again, you will notice that the change directly influenced the link properties of the current link and the link located after the buffer module. Moreover, you can see that the buffer module is now marked red. That's because we now have created a so called "illegal condition". This means, one or more of the link properties are not compatible with the parameter settings in the module. In our case, the parameter settings of the ImageBuffer cannot be used with the settings of the link. If you close all property windows and perform a design rules check **Ctrl+F7** you will get error messages as shown in the following figure.

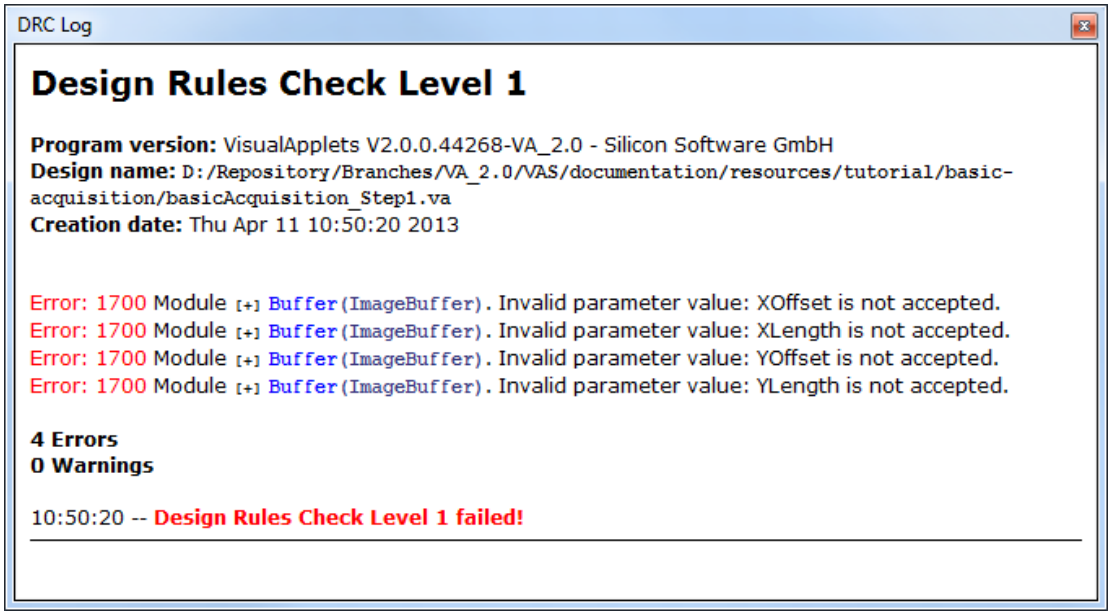



Figure 9.5. DRC Error Messages Invalid Parameters

You can click on one of the error messages to highlight the module with the error. This is very useful if you have large designs in multiple hierarchies and cannot immediately see where the operator is located. Now, open the properties dialog of the Buffer module. Again, you can see that the four parameters which cause the error are dyed in red.

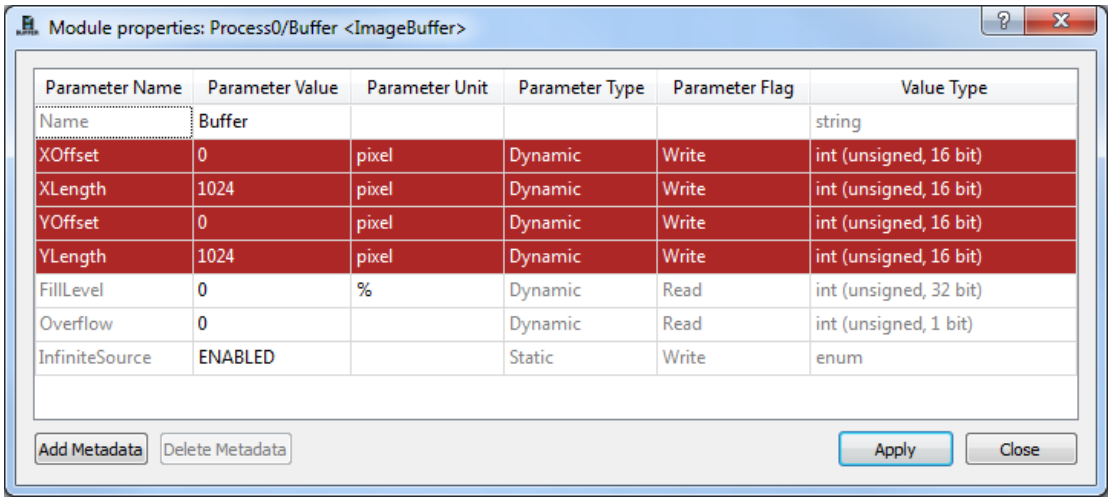



Figure 9.6. Red Parameters show Illegal Condition

Also, the reason for the error now becomes obvious: A region of interest in the operator is set which is larger than the maximum allowed image dimension on the link. This will violate the operator's parameter range and the VisualApplets design rules and therefore, causes the error. You can immediately bring your design into a valid condition by changing the XLength and YLength parameters to value 512.

We are almost done with our design. We can already start the build and use the design in hardware. However, there is one more step we should consider. So far, we connected our 12 bit pixel values directly to the DMA module. This will cause the transfer of 12 bit for each pixel to the PC. This format is very difficult to be handled by software programs as pixels are distributed to multiple bytes. Therefore, each pixel should consume exactly one or more byte. In our example, we expand our pixel to 16 bit. To do so, we need to place another operator between the buffer and the DMA module. There exists several operators to solve our task. We will use operator *ConvertPixelFormat* from the base operator library. Place the operator between the buffer and the DMA module. If you place the operator directly over the

link, the operator will automatically be inserted. Next, you should rename the module to "To16Bit" and open the link properties behind the operator. As you can see, the default link bit width is chosen for the module output which is 8 bit per pixel. Thus, the module has overwritten the input link bit width. Any changes of the input link bit width are ignored. The module will always output the parameterized value. The default eight bit per pixel will decrease the bit width from 12 to 8, but instead, we want to increase the bit width. So, you just need to change the link property value to 16 bit and confirm the changes.

The ConvertPixelFormat operator will now expand the 12 bit input pixels to 16 bit by inserting four constant zero-bits at lower positions. In other words, we performed a left shift by four bit. Instead of using the ConvertPixelFormat operator, operator ShiftLeft can be used in alternative. Delete the ConvertPixelFormat operator from you design by selecting it with the mouse and press delete from the pop-up menu or from the main menu bar; or simply press **Del** . Insert operator ShiftLeft from the arith operator library. Insert it into the design and check the link properties. As you can see, the link bit width is still at 12 bit and cannot be changed in the link properties. For this operator, the link bit width is changed by a parameter. Namely parameter Shift. Change the parameter value to 4 and, apply the changes and go back to the link properties. As you can see, the link bit width has now been changed to 16 bit. To summarize: Some link properties can be changed on the link while some are changed using operator properties. In the respective example these two methods are required for the operators because one is used to specify the output bitwidth independent of the input bitwidth i.e. the shift value is automatically determined. While the shift operator is used to specify the shift value. Here, the output bitwidth is automatically determined by the parameterized value and the input bit width.

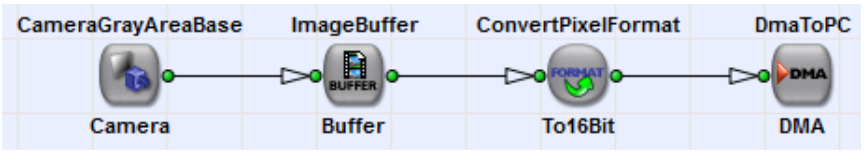



Figure 9.7. ConvertPixelFormat Operator Added for 16Bit Output

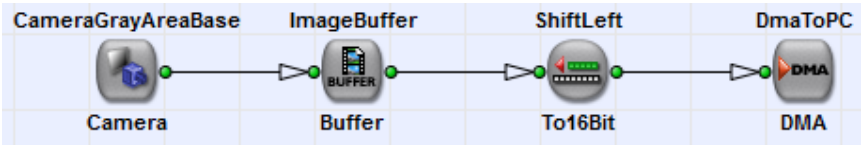



Figure 9.8. ShiftLeft Operator Added for 16Bit Output

In the example, we inserted four bits at the lower bit position. If you rather want to add four bits to higher bit positions to get 16 bit you can use operator *CastBitWidth* instead.

An explanation and examples of the bit manipulation operators can be found in the operator reference. See Section 20.8, 'ConvertPixelFormat', Section 19.15, 'ShiftLeft', Section 19.16, 'ShiftRight' and Section 20.2, 'CastBitWidth'.

We now have finalized the extension of our basic acquisition design. Next, you can build and use your design in hardware. You have learned on how to set parameters and how to solve conflicts. In the next section, a list of the basic acquisition applets for all kind of cameras is given.

## 9.2. Multiple DMA Channel Designs

Most VisualApplets applications require the acquisition of camera images, process the image through image processing algorithms and output the results to the host PC. Besides that users want to monitor the original monitoring images. Thus we need two image outputs. The original monitoring images and the processed results of the algorithmic implementation. In the following we will implement a VisualApplets design which fulfills the following specification:

- grayscale images using a Camera Link camera in base configuration mode
- maximum resolution = 1024 x 1024 pixel @ 8Bit/pixel
- buffer images and forward every 10th acquired image to the host PC (monitoring)
- binarize each of the acquired images uising thresholding and output the resulting images to the host PC
- the binarized output images have to be in the format: 0 = black, white = 255 (8Bit/Pixel)

Using the specification, we can generate a block diagram.

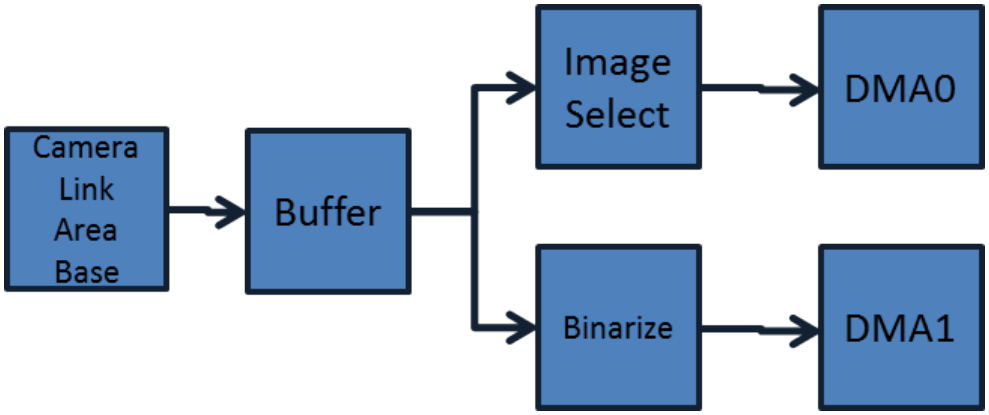



Figure 9.9. Block Diagram of Threshold Binarization Design with Monitoring

As we can see images are grabbed and buffered. After the buffer the processing pipeline is split into two paths. The first path is responsible for the output of every 10th image for monitoring. The second path includes the binarization and the output of the binary images.

Most VisualApplets implementations follow the same principle: Image acquisition and buffering comes first. After the buffering, the image processing logic is located. In Section 3.5, 'Data Flow ' you can find more information on VisualApplets processing pipelines and the basic setup of designs. In the following we will go through the implementation of a VisualApplets design which fulfills the specification step by step. Besides the VisualApplets implementation, we will verify the design using the build-in simulation and use the design in real hardware.

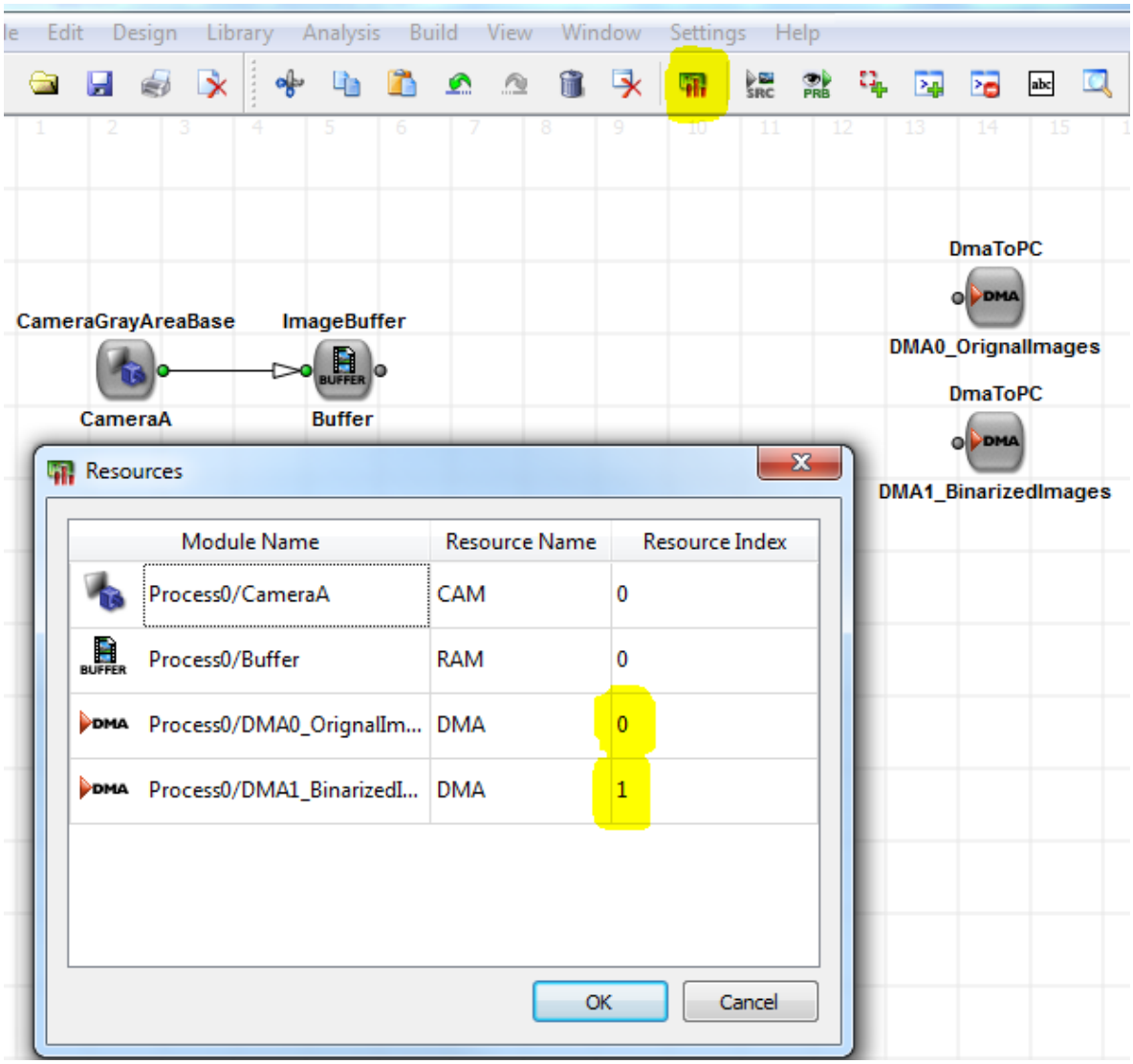
### 9.2.1. VisualApplets Implementation of Binarization with Monitoring

1. We will start with the basic operators we used in the previous examples. These are the camera operator *CameraGrayAreaBase*, the buffer *ImageBuffer* and the operator to transfer the results to the host PC, namely *DmaToPC*. In contrast, this time we will need a second *DmaToPC* module in the design to transfer the original image as well as the binarized results. Locate and rename all operators as shown in the following design.



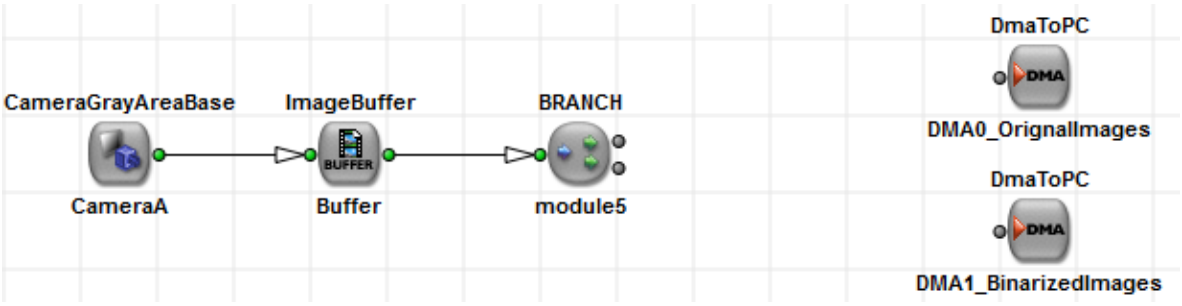

2. As you can see, we renamed the DMA modules to DMA0_OriginalImages and DMA1_BinarizedImages. Thus we gave them an index. That's because each DMA channel uses a channel index. This channel index is used to access the data from any PC software. Most non FPGA-internal resources use an index to specify their location. Some of the operators using FPGA external hardware are the camera operators, operators using DRAM e.g. the ImageBuffer, operators using general purpose input outputs e.g. trigger signals and DMA operators. In fact, all operators which are in our design so far are using external resources and can be allocated to an index. The camera operator can be allocated to the Camera Link port A or B. The buffer can be allocated to one of the four DRAMs on the microEnable IV VD4-CL and the DMA to one of the DMA channels. A detailed explanation of the use of the device resources can be found in Section 3.8, 'Allocation of Device Resources'. A list of the available device resources for all supported hardware platforms is presented in Appendix A, '*Device Resources*'.

   Most of the device resources can be changed using the resource dialog window in VisualApplets. Some can change their resources using the operator's parameters. To change a device resource, we have to open the resource dialog window which can be accessed from **Design** -> **Resource** or by using the icon in the edit menu bar.




Ensure that the DMA module DMA0_OriginalImages is allocated to index 0 and DMA1_BinarizedImages is allocated to index 1. The camera module should be allocated to index 0 which represents Camera Link port A. We don't really care which of the DRAM chips our buffer is using, so we let VisualApplets choose an index automatically if there is a conflict between some memory operators. In general, VisualApplets will always chose an unused index. If you have two camera modules and delete the module using index 0, the remaining camera module will use index 1. So ensure to check all device resource indices before the start the build of the applet. The resource indices cannot be changed after synthesis.

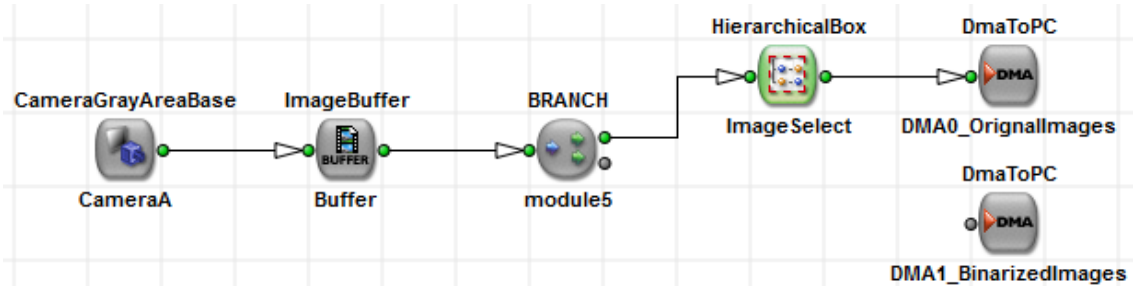
3. Next, we need to include our image processing algorithms. As we have to implement two different image processing algorithms, we need to branch our pipeline into two paths. This is simply done by a branch operator. Locate the operator in the base operator library and drag it into your design. When you release the mouse button a window will pop-up which will allow users to specify the number of output links i.e. the number of branches. Select two and OK. You will now have a branch module in your design with two output links and one input link. Move the module next to the buffer and connect them.




We now have branched our pipeline and can implement both image processing algorithms between the branch and the DMA modules.

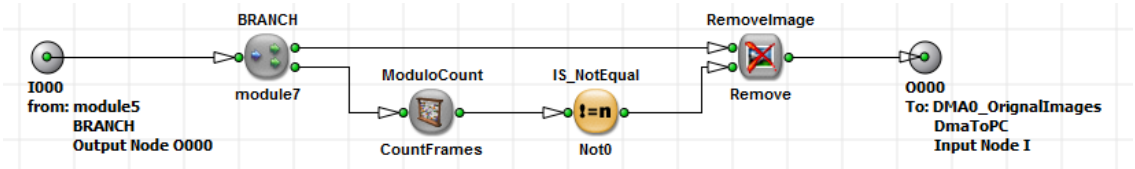
a. First, we will implement the original image output path where only every 10th image is required to be output. Instead of placing all required operators on the main design level, we will use a hierarchical box. With the help of hierarchical modules, it is practical to implement a hierarchical structure into your design and combine an arbitrary number of operators to a new operator. The use of hierarchical modules ease the overview of the construction area. Insert an hierarchical box with one input and one output into your design. You can find the hierarchical box in the base operator library, from the design area pop-up menu or form the main menu "Design".

   Place the HierarchicalBox module between the branch and the DMA and connect them. You can rename the module using it's context menu or by selecting it and hitting the F2 key.




   To edit the hierarchical module content just double-click on it. A new blank design window will open. It only includes the input and output connectors. We will place the required operators in-between and connect them to the ports. To exit the hierarchical module hit Backspace or icon .

   To remove 9 out of 10 images, we need the operators as shown in the following.



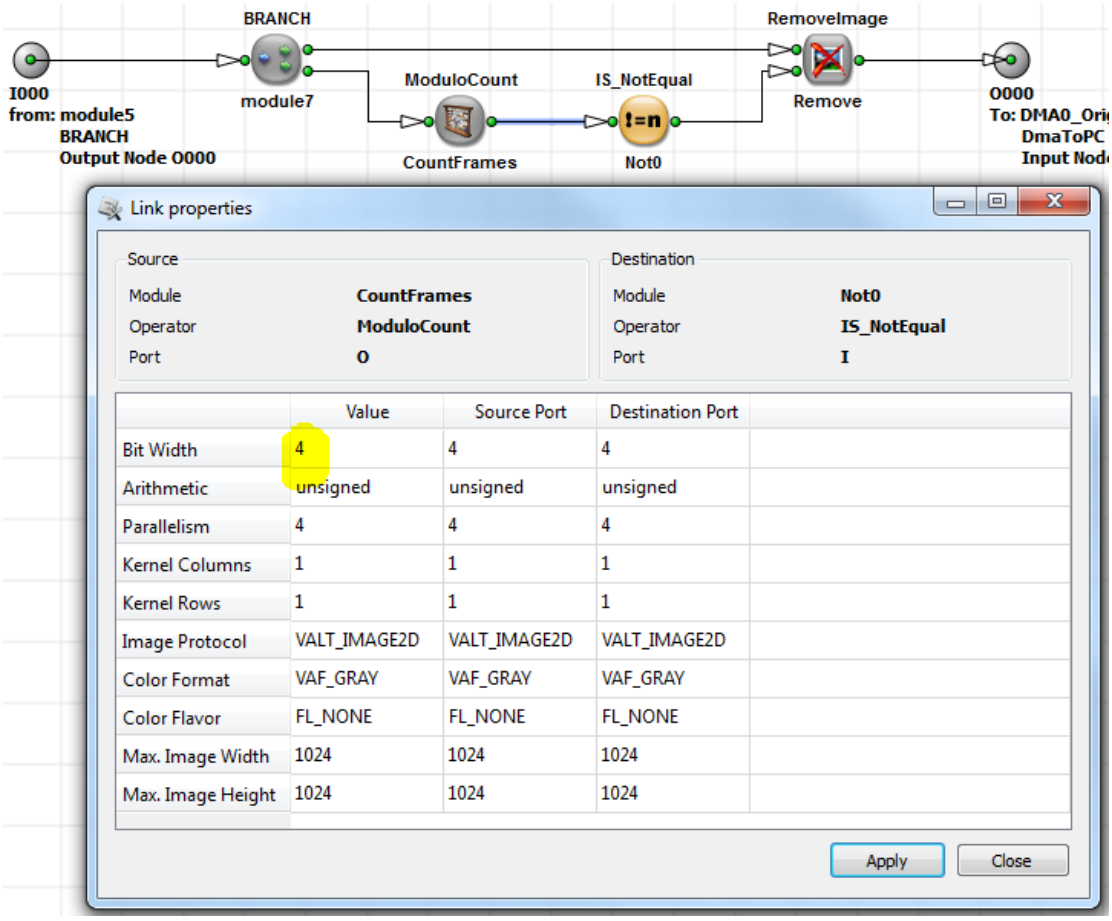

   and parameterization:

   • Properties of module CountFrames

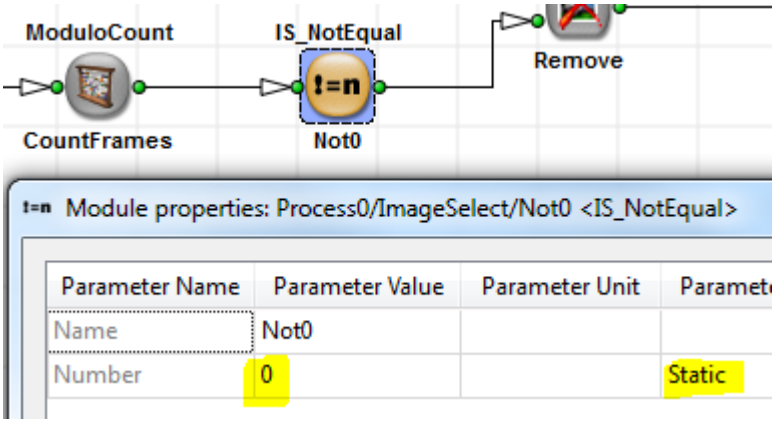
Module properties: Process0/ImageSelect/CountFrames <ModuloCount>

| Parameter Name | Parameter Value | Parameter Unit | Parameter Type |
|---|---|---|---|
| Name | CountFrames | | |
| Divisor | 10 | | Dynamic |
| CountEntity | FRAME | entity | Static |
| AutoClear | NONE | mode | Static |

- Link Properties of the output of module CountFrames

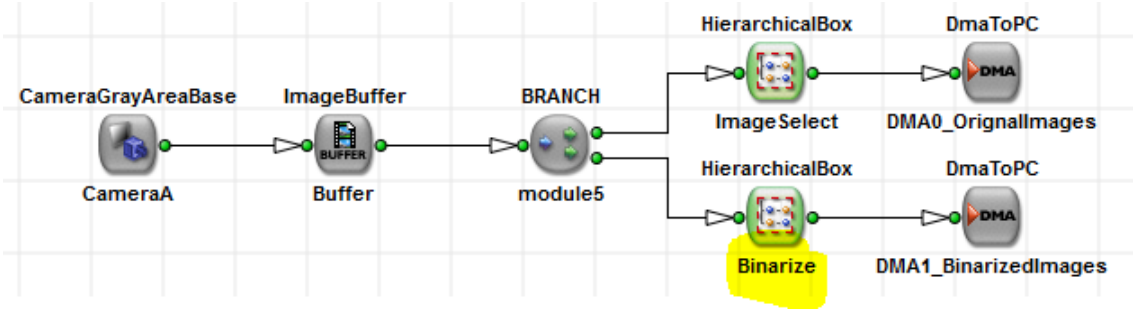



- Properties of module Not0

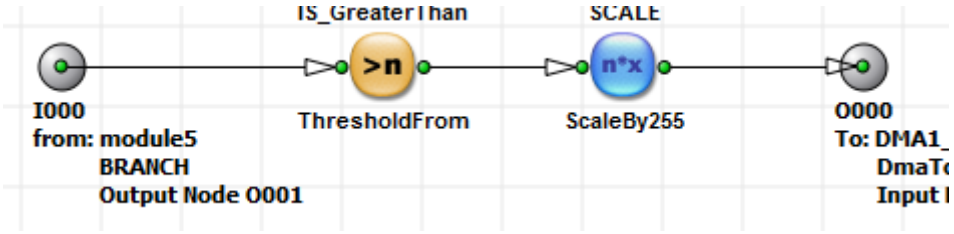



How it works: The CountFrames module will replace the pixels of each image by a value between 0 and 9. Hence, it will repetively count from 0 to 9 with each image. Module Not0 will replace every pixel of each image by value 1, if the current count value is not 0. The RemoveImage module deletes each image, if the pixel value of the second input "Rem" is value 1. Thus, image number 0, 10, 20, 30, ... will be bypassed. All other images will be deleted.

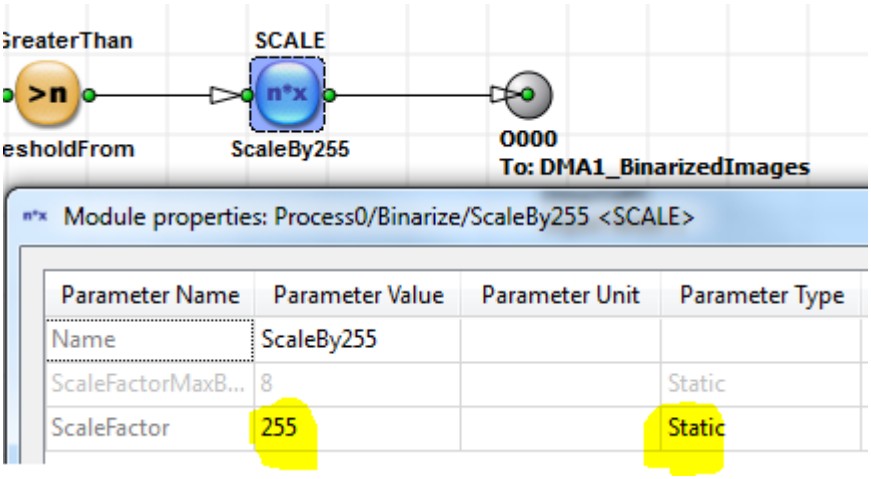
b. For the implementation of the binarization we will add another hierarchical module into the design.




The operators in this hierarchical module are "Is_GreaterThan" and "Scale".




Module ThresholdFrom represents the actual binarization. For each input pixel value grater than value specified by parameter Number, value one is output, otherwise zero. Thus, the output bit depth of the module will be one bit. You can check this by opening the link properties. As we do not want to output one bit values, we need to bring our results to eight bit values. A very simple solution to do is to scale each of the values with 255. Thus, binary value one becomes 255 and binary value 0 remains at 0. Set the parameters of the ScaleBy255 module to the following values.




As you can see, we have set the parameter type of parameter ScaleFactor to Static. For this operator this is very important as it will require much less FPGA resources compared to a dynamic parameter. In general, always set parameters to a static type if you know that you will not need to change them after synthesis.

4. The final step of your implementation will be the check for design rule errors. Perform a DRC to check for errors. If you have no errors you can now build your design and use it in hardware. In alternative, you can verify your implementation using the simulation which is shown in the next section. Do not forget to save your design from time to time.

### 9.2.2. Verification of the Binarization Design using Simulation

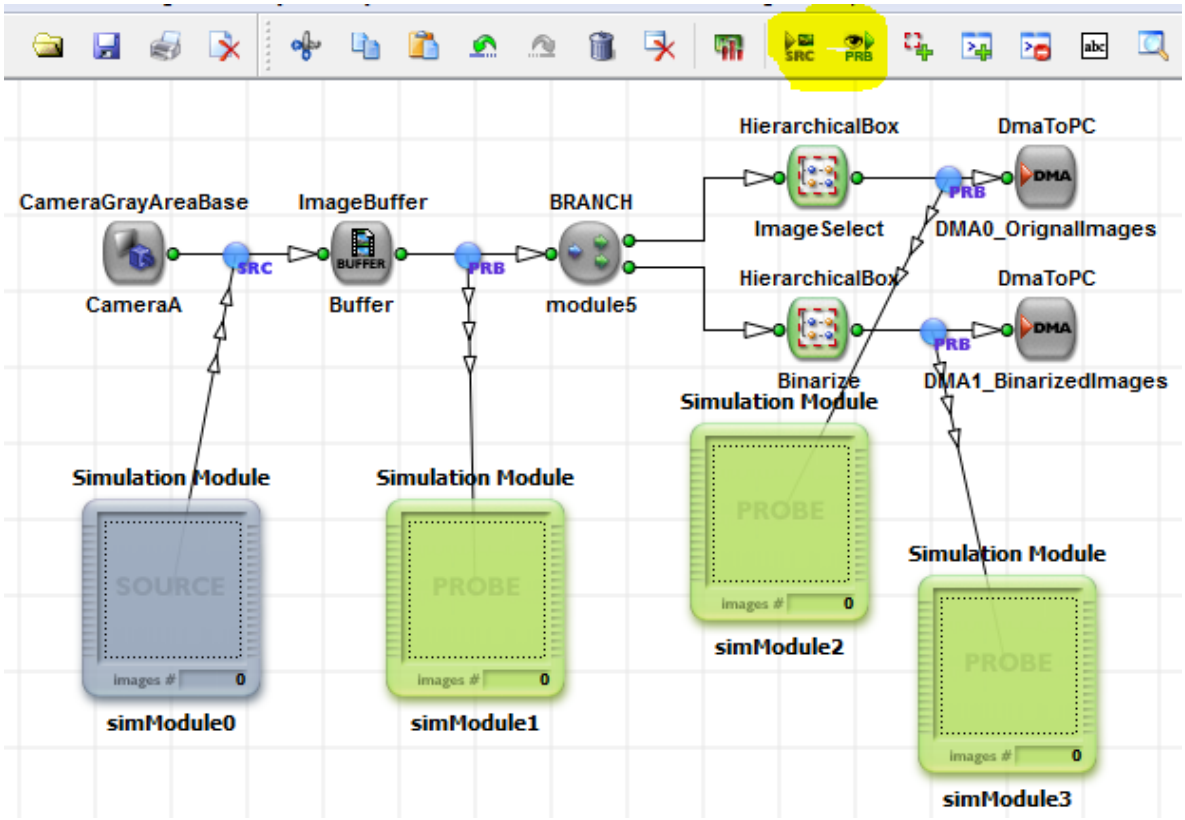
For most designs it is very useful to verify the functionality of the implementation using the simulation before the use in real FPGA hardware. We will verify our design using the simulation in the following.

1. **Preparation**

   First we will need an image to simulate. You will need to use an image file in TIFF or BMP format. You have to use an image with a single color component if you like to inject the image data to an image processing pipeline using grayscale images as in our example. More information about allowed image files for simulation can be found in Section 3.10, 'Simulation'. There are some sample images in the VisualApplets installation folder.

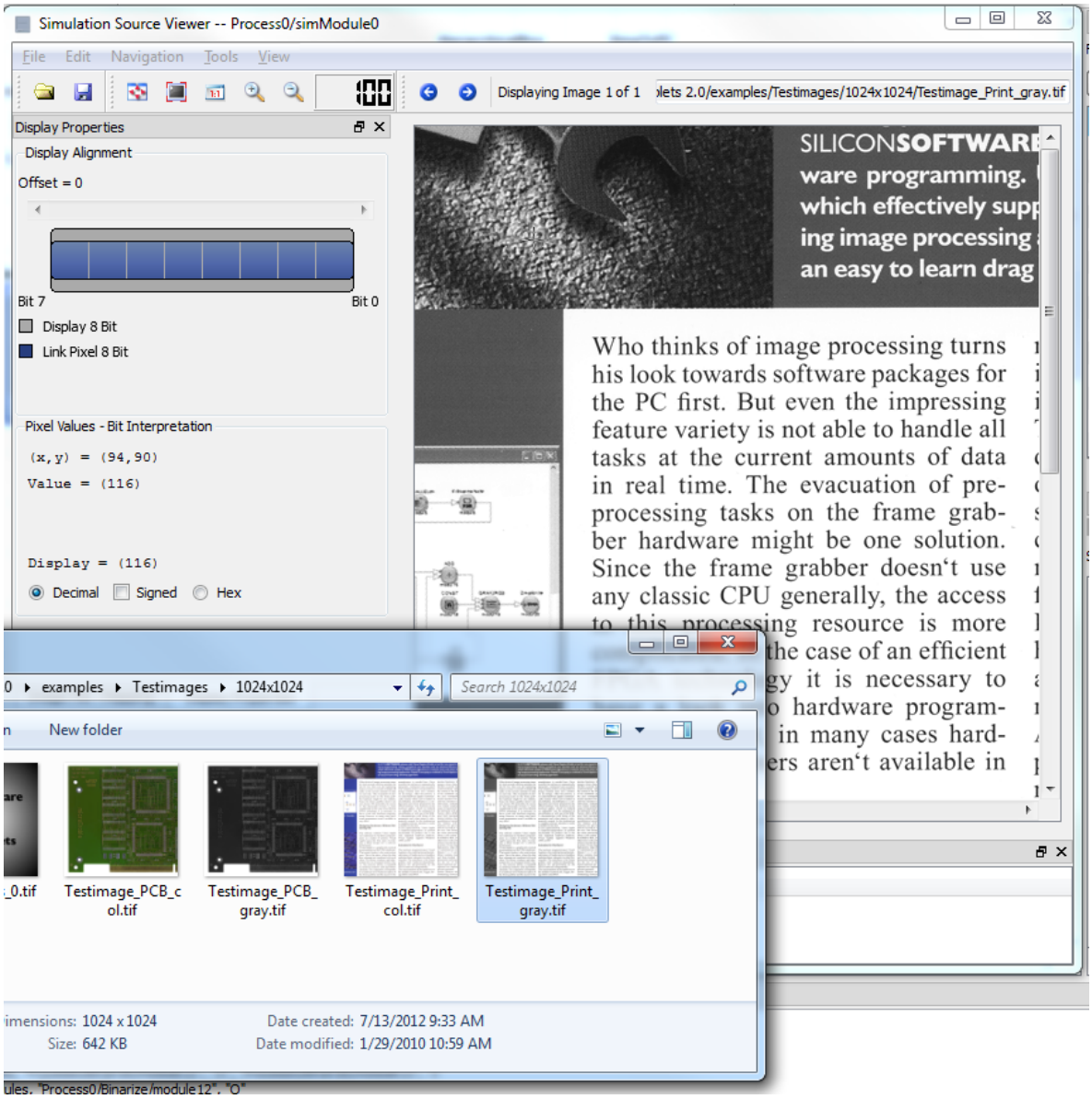
2. **Sources and Probes**

   Image files can be injected to any image transporting link using "simulation_source"" modules and can be monitored on any other link using "simulation_probe" modules. In our example, we add a simulation source to the link between the camera and the buffer and simulation probes according to the following screenshot. You can add simulation sources and probes from the "Analysis" menu, from the context menu of the design window, or simply from the icons marked in the Figure.




3. **Add Image File**

   To add an image file to the simulation source, open the Simulation Source Viewer using a double-click on the source in your design. Now simply drag-and drop your image file to the Simulation Source Viewer window or use the **File** -> **Open** menu.




The image file was now added to the simulation source. You can add more files to the source by simply dragging them on the module or window. At the bottom of the Simulation Source Viewer Windows you can see all images in the sequence.

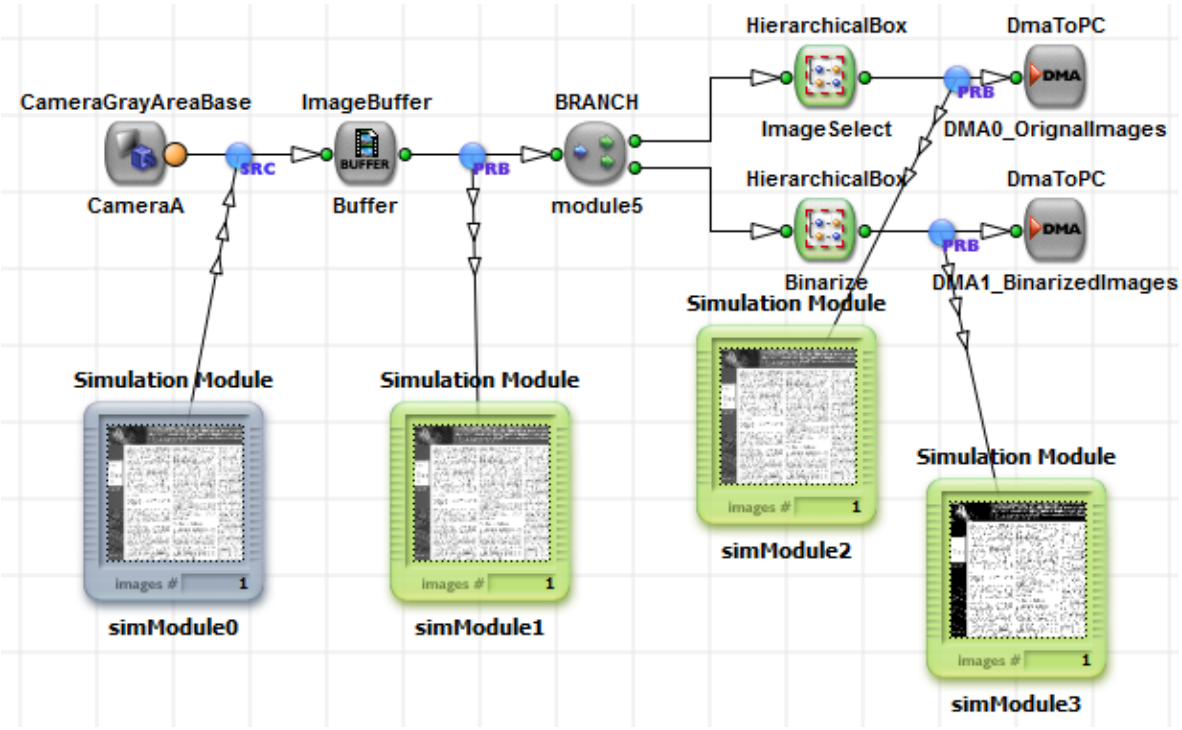
4. **Simulation Start**

   We are now ready to start the simulation. Select **Analysis** -> **Start Simulation** ( **F9** ) to open the main Simulation window. Each time you open this window a design rules check will automatically be performed. It is not possible to simulate a design with a failed design rules check.

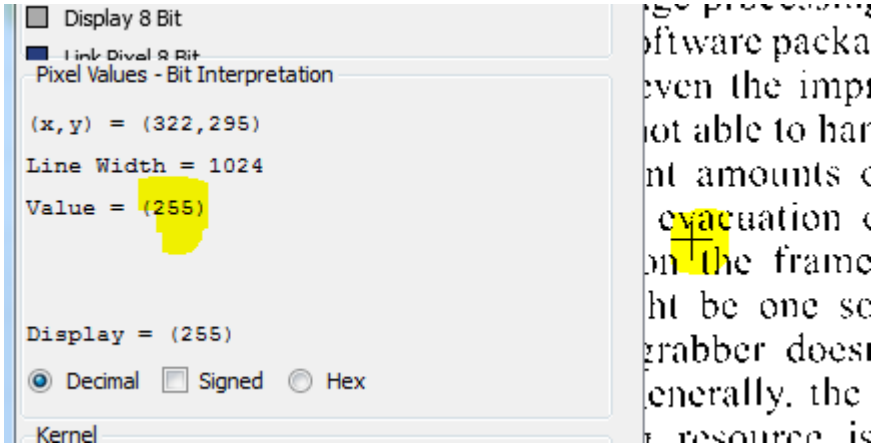
   In VisualApplets, simulations are processed in cycles. Each cycle represents the injection of an image from all simulation sources in the design. Simply click on **Start** to start the simulation. VisualApplets will now perform the simulation. Depending on the complexity of the design this might take some seconds.

5. **Check Results**

   We can now check the results of our simulation. Close the Simulation window. The simulation probes should now have been filled with results.




If you open the Simulation Probe Viewer of the probe located behind the binarization, you can immediately see if the simulation worked. The image should include pixels having values 0 and 255.




Results can now be saved to disk, if required.

6. **Sequence Simulation**

We could verify the binarization in the previous step. However, a single simulation image is not sufficient to verify the monitoring output as only every 10th image is supposed to be transfered to the host PC. Thus we need to simulate a sequence of images.

a. Open the Simulation window once again.

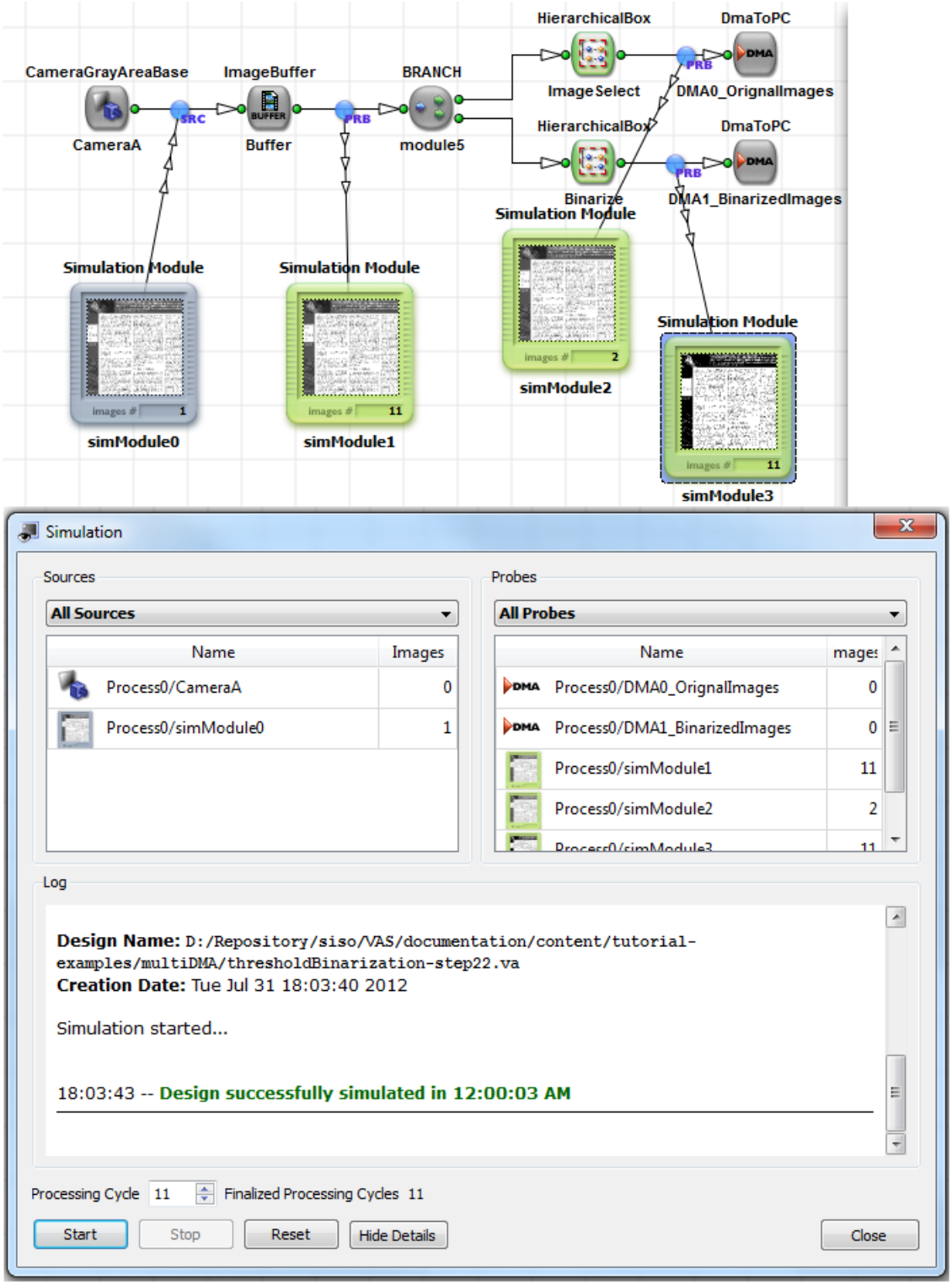
b. Click on **Reset** to reset the sequence from previous simulations. In general, if you want to start a new simulation, it is useful to always reset the simulation to avoid user errors.

c. Click on **Start** to simulate one image. We will get a result in each of the simulation probes. It is not required to close the Simulation window now. You can get the current number

of images in a simulation probe by looking at the numbers of the modules in the design window.

d. Now perform another cycle by a click on **Start**. This time, the image number of all simulation probes should have been increased to two except the one at the DMA0 module.

Repeat this step until you have simulated 11 cycles in total. The probe at DMA 0 should now include two images. This shows the correct functionality of our implementation.



e. Instead of clicking 11 times on **Start** you can set the number of processing cycles using the spin box in the Simulation window. The simulation result will be the same. Also, you can set 10 simulation cycles, start them and start the next 10 cycles after. As you like...

### 9.2.3. Verification of the Binarization Design in Hardware

After we verified the functionality of our implementation we can test and use it in hardware. You will need to build your design to get the hardware applet file HAP. (User Manual: Section 3.12, 'Build') Ensure that your design matches the hardware platform e.g. mE4VD4-CL in our example. Moreover ensure that your project is using the correct architecture e.g. Windows 64 Bit. It has to be the same as the installed Framegrabber SDK. Start the build by **Build** -> **Synthesize** ( **F7** ). VisualApplets will now use the FGPA manufacturer tool to translate your design into the FPGA bitstream.

The result of the synthesis is an HAP-file. It contains the FPGA bitstream and the software interface to access data and parameters. The HAP file is located in your Framegrabber SDK installation directory in subdirectory "Hardware Applets" or, if no Framegrabber SDK is installed or you selected another target runtime, the file is located in the folder specified in the system settings (See Section 4.7, 'System Settings').

We will test our design in the Framegrabber SDK tool microDisplay. microDisplay is a tool for a first test of applets. Open microDisplay and load the applet. microDisplay will show two DMA image windows, namely the original image window where only every 10th image is shown and the second DMA window showing the binarized images. A detailed explanation on the use of applets in the Framegrabber SDK can be obtained from the Framegrabber SDK documentation [https://docs.baslerweb.com/frame-grabbers/managing-applets-micro-diagnostics].

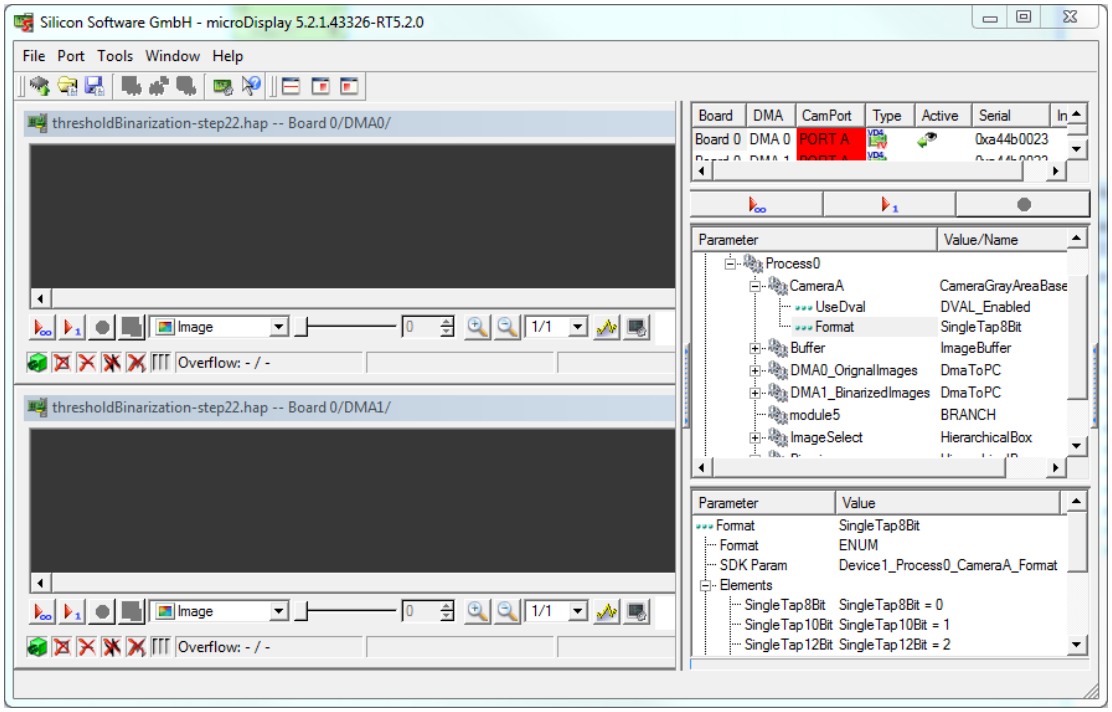



Figure 9.10. Use of the Binarization Applet in microDisplay

# 9.3. Synchronization of Asynchronous Image Pipelines

## 9.3.1. Synchronizing Cameras

A powerful feature of VisualApplets is to combine the images of different camera sources. In the following we will learn on how to

1. switch between camera sources
2. overlay the images of two cameras
3. multiplex the camera images
4. stitch the images of two cameras

### 9.3.1.1. Switch Between two Cameras

To switch between two cameras, we need a simple design including two camera modules. In VisualApplets operator SourceSelector (see Section 32.23, 'SourceSelector') can be used to switch between two asynchronous sources. This operator allows the selection of one of it's inputs, while the data on the other inputs is discarded. If one of the inputs is selected, the others behave like they are connected to a Trash operator. It is possible to switch the inputs while the acquisition is running. The operator will switch with the start of a new frame. The frame rate and image sizes of both inputs do not have to be the same.

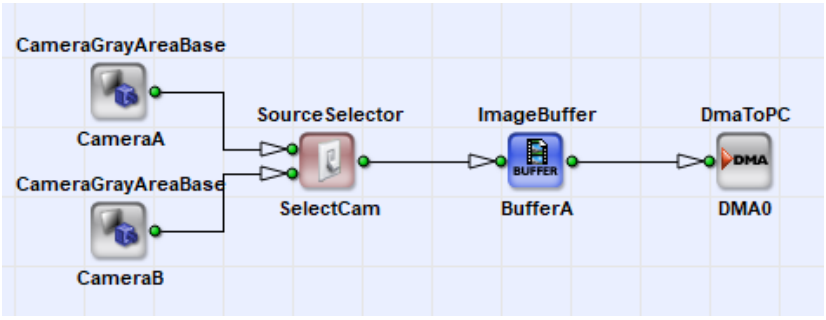



Figure 9.11. VisualApplets design to switch between two cameras

### 9.3.1.2. Combine Image Data From Two Camera Sources - Building an Overlay Blend

In the following, we will have a look on how to combine the image data of both cameras. In our example, we use an overlay-blend to combine both images.

The specification for our example will be:

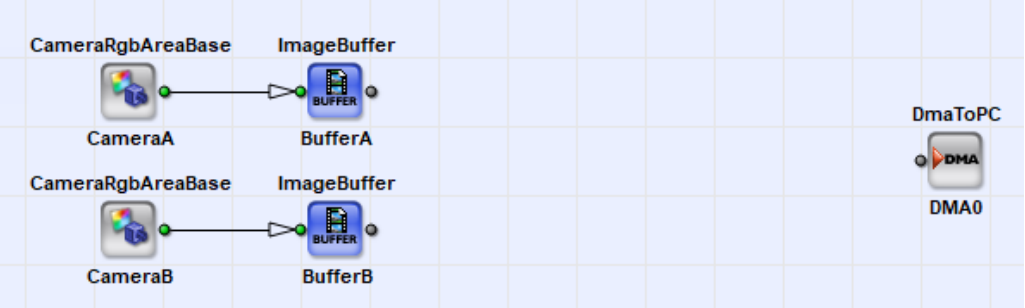
- Camera: Two CameraLink RGB area scan cameras in base configuration mode
- Resolution: 1024 x 1024 pixel
- Overlay-Blend:

$$ResultColor = \begin{cases} \dfrac{2ColorAColorB}{256} & \text{if ColorA} \leq 128 \\ 255 - \dfrac{2(255 - ColorA)(255 - ColorB)}{256} & \text{else} \end{cases}$$

Equation 9.1. Overlay Blend

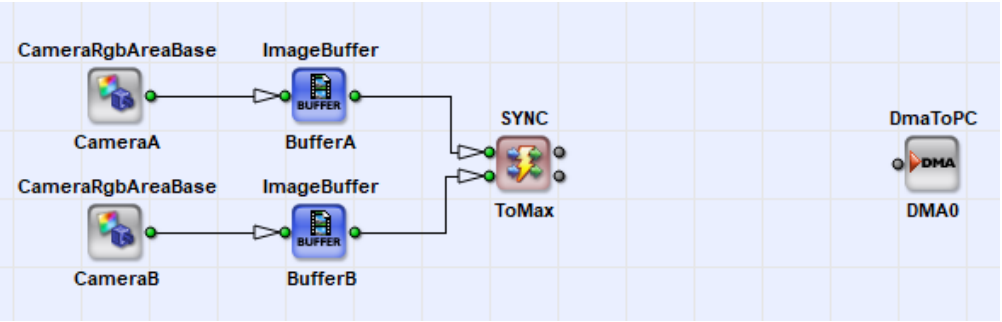
1. **Basic Operators**

   We start with the basic operators in our design as shown in the following figure. We can use all the default parameters and link properties.




2. **Synchronization**

   Next we need to synchronize the data streams of both cameras. This can simply be done by using operator *SYNC*. The *SYNC* operator controls it's inputs so that the outputs are fully synchronized. As you can see, we used two buffer modules before the synchronization. We need to use them because we cannot ensure that both cameras are 100% synchronous.




   Although we synchronized our image data, we still need pay attention to some constraints:

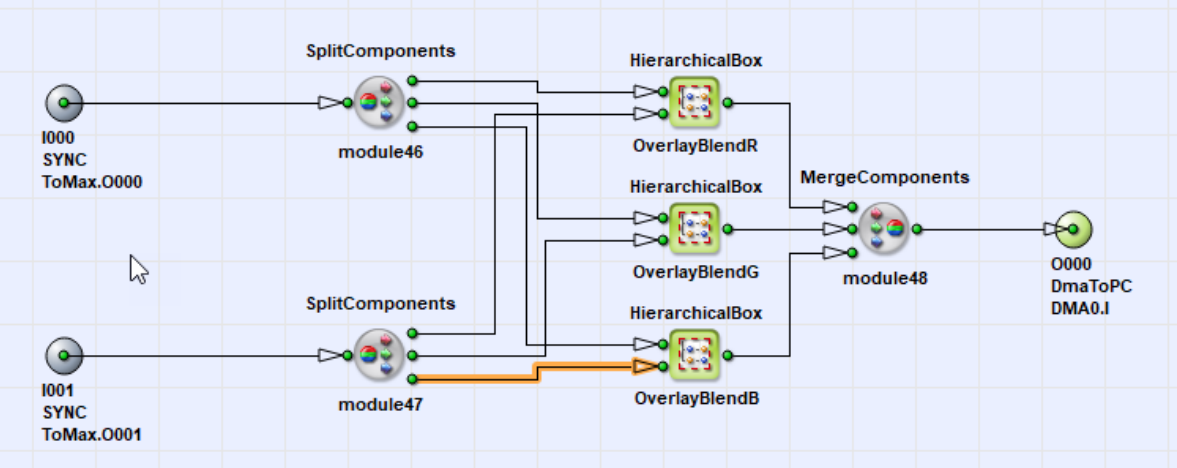
   - The number of frames i.e. the frame rate of both cameras has to be the same. If one camera has a higher frame rate than the other camera, one of the ImageBuffer modules will need to store some images and might get in an overflow condition. Thus, for a long time acquisition, both cameras have to acquire their frames using the same frame rate. A trigger system can ensure this.

   - The image size and the image position of the ROIs of both cameras may differ. The *SYNC* operator will, depending on it's parametrization, automatically expand the smaller image to the larger one or cut the larger image. See the description of the *SYNC* operator for more details: (Section 32.26, 'SYNC')

3. **Overlay-Blend**

   The last step to do is to implement the overlay blend to combine both images. For convenience, we use a hierarchical box for the overlay-blend implementation.

   The overlay blend has to be individually applied to all color components. Thus we need to split our links into all three color components. The overlay blends are placed in hierarchical boxes once again. Thus we have three hierarchical boxes inside the current hierarchical box. You can use copy and paste to duplicate the boxes for each color component.




We will have varying possibilities to realize the equation given above using VisualApplets operators. In general, you should always try to adapt and optimize an equation for FPGA use before implementing it.

The equation includes some multiplications and divisions with constant values. These operations are available in VisualApplets but cannot be very efficiently implemented on FPGAs. If the multiplication or division can be written as a power of two multiplication we can use a simple bit shift for implementation. A bit shift will require no resources at all! We will simplify

$$2/256$$

to

$$1/128 = 1/2^7$$

which is a simple right shift by 7 bit i.e. we discard the seven least significant bit.

$$ResultColor = \begin{cases} \dfrac{ColorAColorB}{128} & \text{if ColorA } \leq \text{ 128} \\ 255 - \dfrac{(255 - ColorA)(255 - ColorB)}{128} & \text{else} \end{cases}$$

Next, there are some parts in the equation like 255 minus ColorA. This is a simple color inversion. White becomes dark, dark becomes white. An inversion of an value can also be made using a simple inversion of each bit. This allows us to finally rewrite the equation:

$$ResultColor = \begin{cases} \dfrac{ColorAColorB}{128} & \text{if ColorA } \leq \text{ 128} \\ \overline{\left(\dfrac{\overline{ColorA}\ \overline{ColorB}}{128}\right)} & \text{else} \end{cases}$$

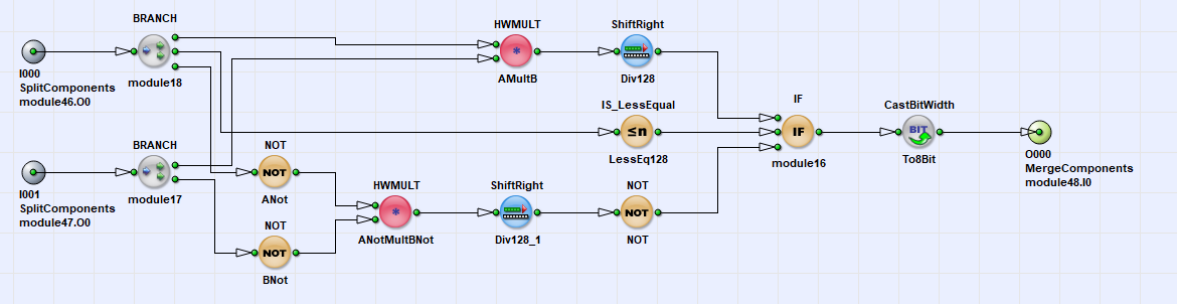
Equation 9.2. Optimized Overlay Blend for FPGA Implementation

We can now start our implementation. The remaining operations in the equation are available as VisualApplets operators. We will need

- *IF*
- *NOT* for the inversion
- *ShiftRight* for the division by 2^7
- *HWMULT* for the multiplication

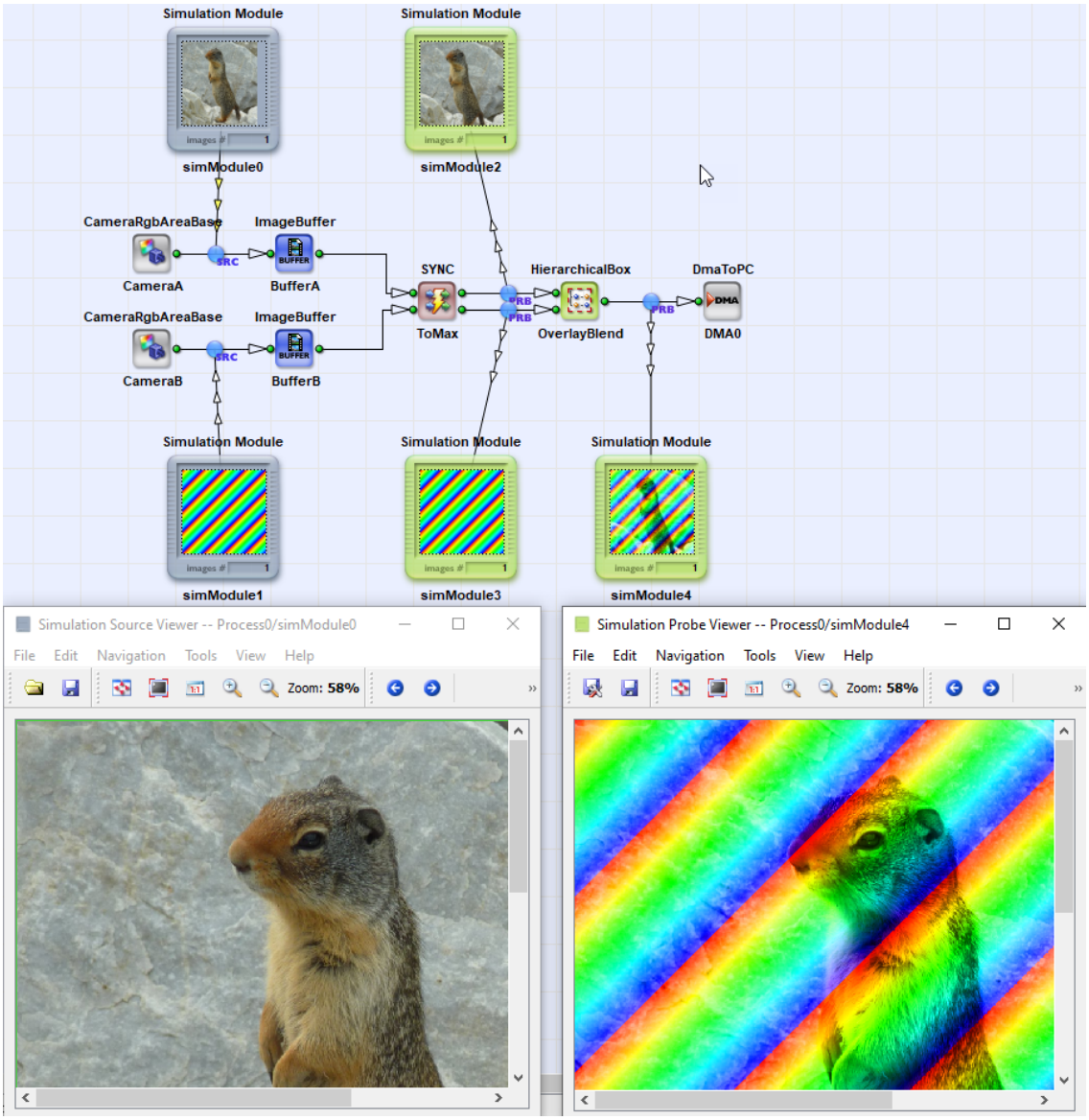
  Besides *HWMULT*, VisualApplets includes the *MULT* operator. The do the same, but *HWMULT* will use dedicated multiplier in the FPGA and therefore, will require much less resources.

- For bit depth limitation, we will need a *CastBitWidth* operator. This operator can be used to switch from 9 bit to 8 bit by discarding the MSB. From the formula we know that there will be no values greater than 255.




4. **Verification using the Simulation**

   a. We can easily verify the functionality of our overflow-blend using the VisualApplets simulation. Use two simulation sources to inject an image for both cameras. Place simulation probes to any link you like. The simulation results will show you the overlay of both images. If interested, you can also check the intermediate results of the overlay in the hierarchical boxes.




b. The verification of the synchronization is not possible with the simulation as the simulation will only reflect the functional parts of the applets and not the timing. A verification in hardware is required.

5. **Adding a Trigger**

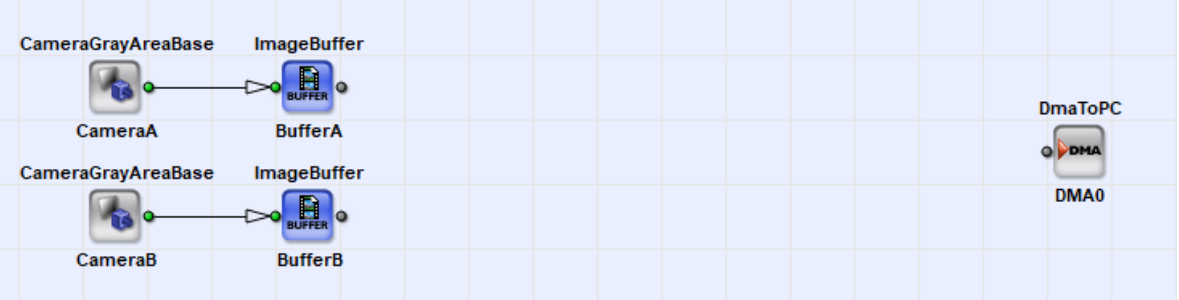
   As mentioned, both cameras have to run at the same frame rate to avoid an overflow on one of the buffers. This can be easily solved using a trigger system.

### 9.3.1.3. Multiplex the Images of Two Cameras

The multiplexing of images or lines in VisualApplets is very easy. We will start with the example of multiplexing the images of two cameras i.e. we want to use the same DMA channel to alternately output the images of camera A and camera B.

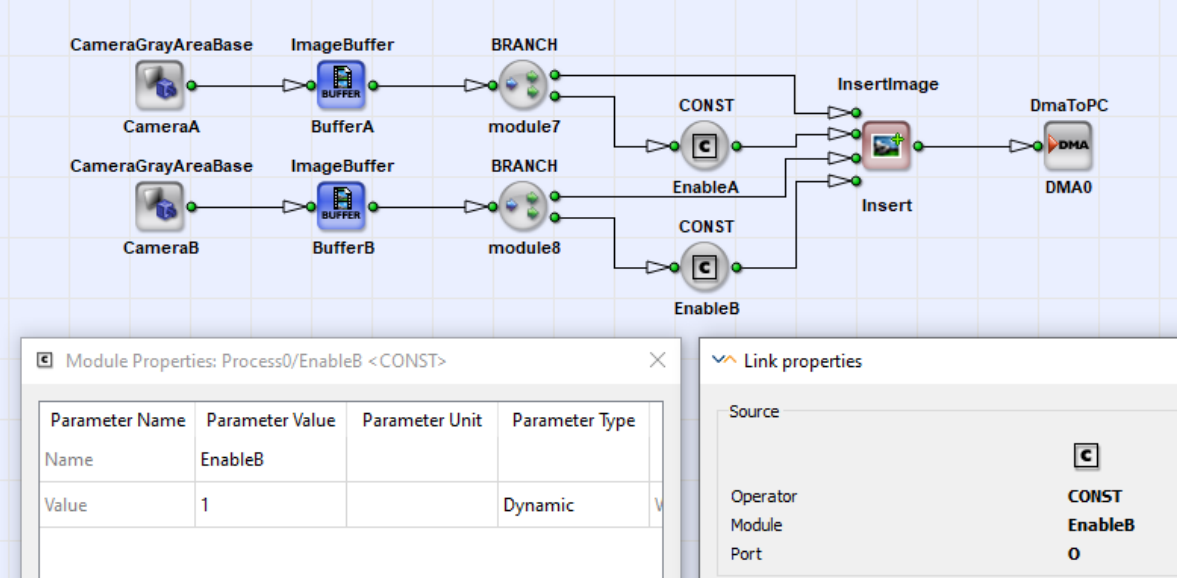
1. **Basic Operators**

   We will need the standard design for two cameras using two cameras modules, two buffers and one DMA.




2. **Multiplex Images**

To multiplex the images, we use operator InsertImage. This operator will forward the images of it's inputs one after another. For example, if we use two inputs, the operator will forward the images of input 0, 1, 0, 1, 0, 1, ... Thus, after one input has been processed, the next one will be used. Therefore, it is required that the number of frames at both inputs is the same i.e. the inputs will have the same frame rate.




As you can see, we placed the image buffer modules before the insertion. This is because the InsertImage operator has no buffer itself. While images of input 0 are forwarded, input one is blocked, thus camera images need to be buffered to avoid the loss of data.

For each input (I0, I1, ..) the operator provides a second input called Ins0, Ins1, ... This input is a control input to control whether images shall be used or discarded. Thus an input can be skipped if value 0 is at the Ins input and will be used if value 1 is provided. The input groups I and Ins have to be synchronous, i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators. See Section 3.6.1, 'Operator Types' for more information. As both inputs are synchronous, they will need to have the same image dimension. InsertImage will use the first pixel on the Ins inputs to decide whether an image is used or discarded. As you can see in the previous screenshots, we simply used a *CONST* operator to enable the use of the images. Set the output link bit depth of the *CONST* operator to one bit and the parameter "Value" to 1 to enable the insertion.

Please note: If the value at one of the Ins inputs is zero, the operator will still need to process the images on this inputs. A value zero does not mean, that no image has to be provided. If no image is provided at an input, the operator waits until an image is available.

As mentioned always one of the inputs is opened at a time, while the others are blocked. The images have to be sourced asynchronously e.g. from different buffers. In the following figure,

you can see two configurations which will not work. You will not get any results from the applet as it is blocking itself. It is a so called "deadlock" condition.

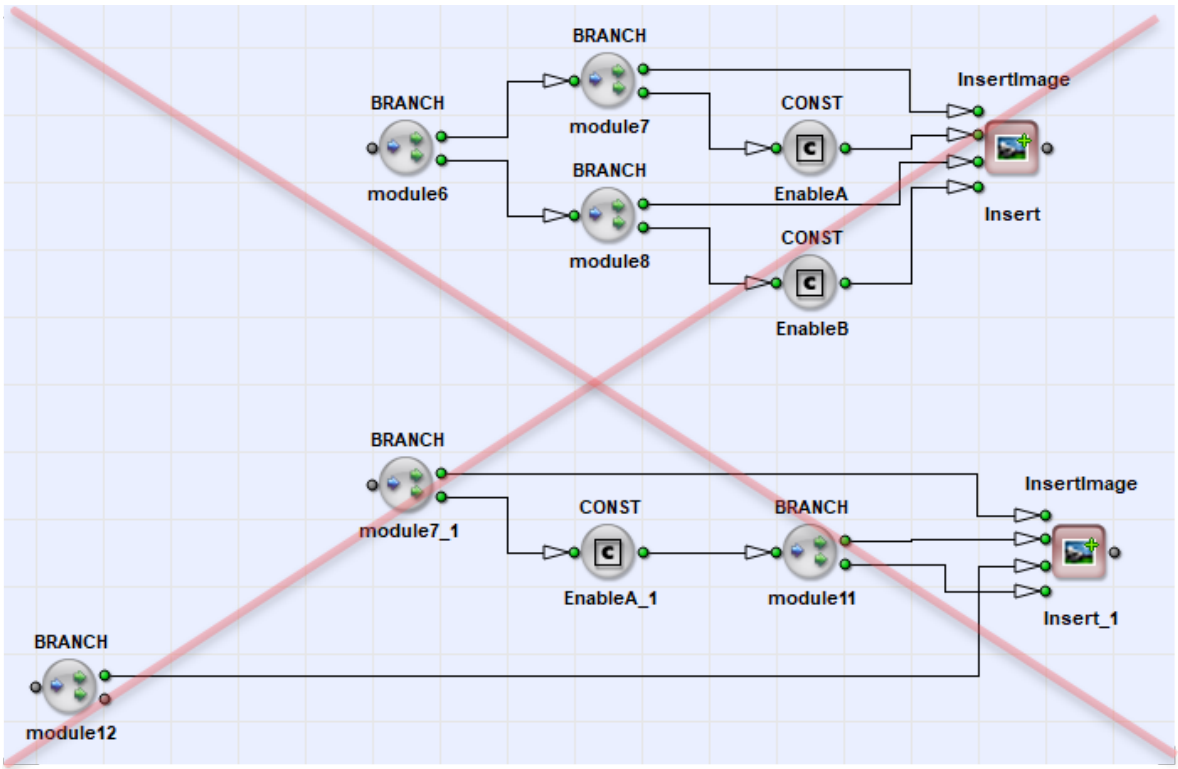
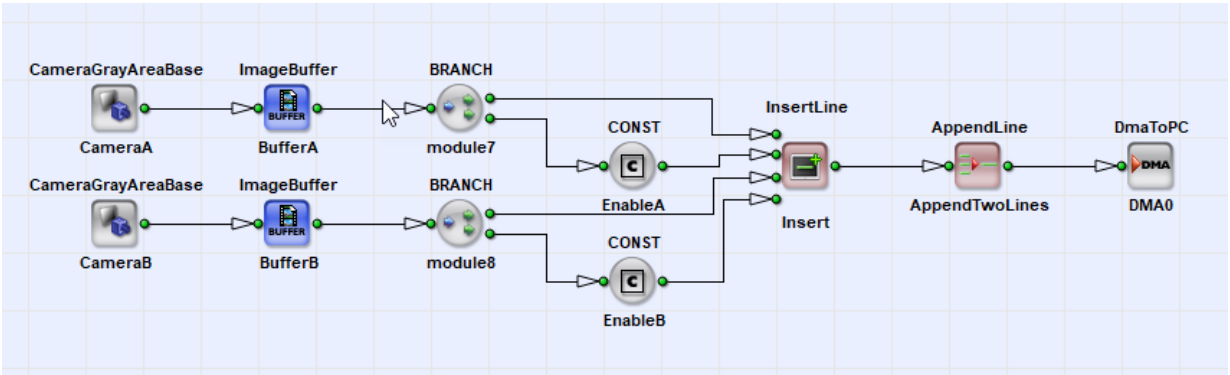



Figure 9.12. Deadlock Configurations using InsertImage

Always check if one of the inputs can be buffered when stopped, while the other one is active. Do not forget that deadlock situations cannot be detected by the DRC nor the simulation. More information on deadlocks can be found in Section 3.6.6, 'Timing Synchronization'.

### 9.3.1.4. Stitching of Two Cameras

In the previous section we multiplexed two cameras images. Stitching is almost done in the same way. We use operator InsertLine instead and need to append two successive lines using an AppendLine operator.




InsertLine will first multiplex the lines of both inputs and after, we need to append two lines.

Of course, it is possible to use the same image source for the last two examples, too. The only thing we have to care is that we have sufficient buffers before the insertion. For inserting an image, we will need to buffer a full image. For duplicating a line, we simply need to buffer one of the lines, while the

same line is processed on the other path. To buffer only a few lines we can use operator ImageFifo. This operators will not use frame grabber DRAM memory. Instead, it uses the FPGA internal BlockRAM memory. We will configure the FIFOs to store a maximum of one line. In the following example, we duplicated an image line, where the duplication is inverted. Thus our result is the original image on the left, and the inverted (negative) version on the right.

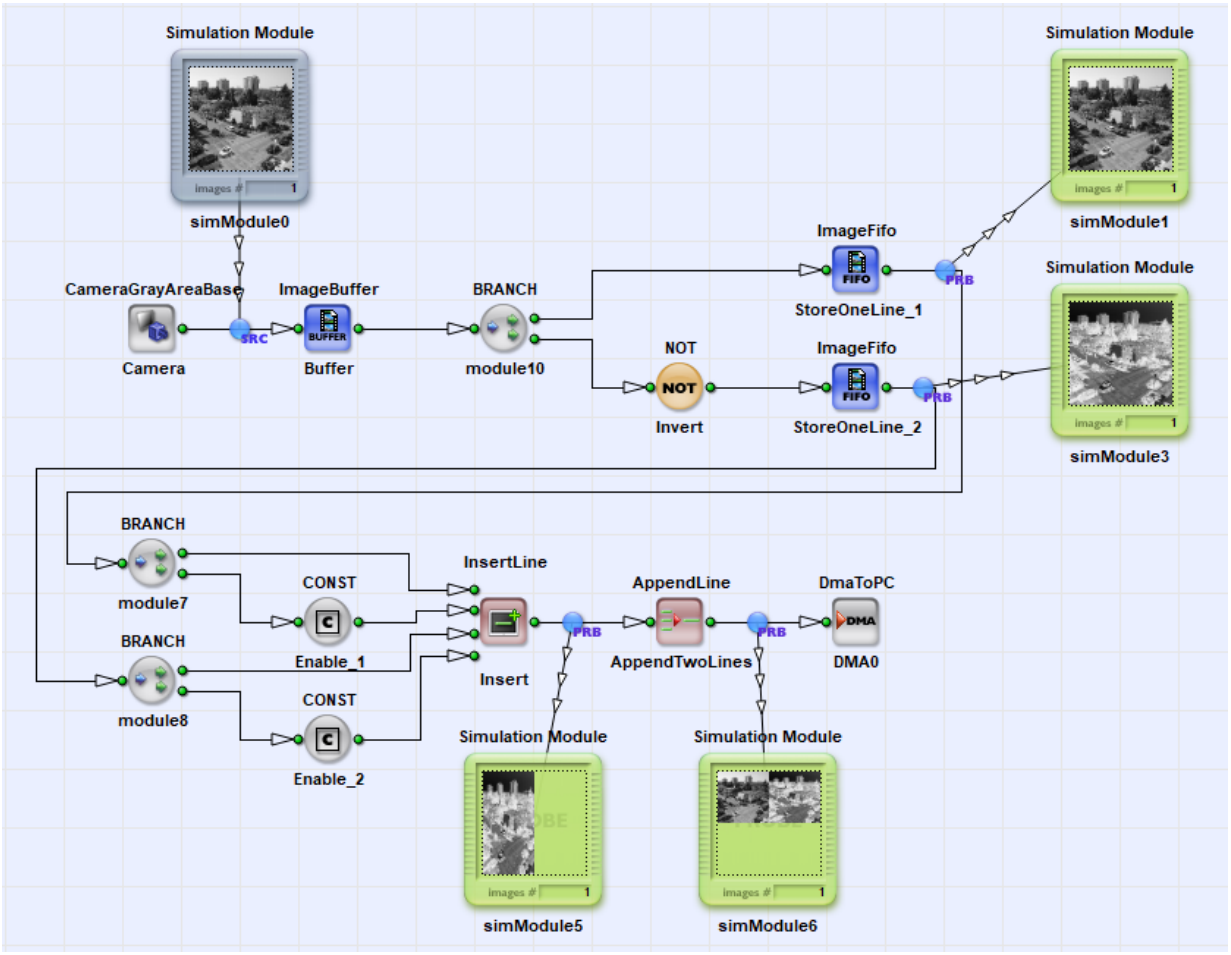



Figure 9.13. Line Duplication

# 10. Basic Acquisition Designs for Varying Camera Types and Hardware Platforms

VisualApplets is designed to easily switch between different camera interfaces and protocols. In the following, basic acquisition designs for different camera interfaces are presented. All designs are implemented for a specific frame grabber. These are the microEnable IV VD4-CL/-PoCL, microEnable IV VQ4-GE, microEnable 5 marathon VCL, LightBridge and microEnable 5 VCX-QP, microEnable 5 VQ8-CXP6D and microEnable 5 VD8-CL platforms. If you like to use another frame grabber which has the same camera interface and is of the same frame grabber generation, you can save the design to another frame grabber model. Check Section 4.5, 'Target Hardware Porting' for more information on how to change the frame grabber of a design.

The following examples will not explain details such as bandwidth and the use of multiple DMAs. We have a look on this in Section 9.2, ' Multiple DMA Channel Designs '.

The following table lists all basic acquisiton examples.

| Interface | Sensor | Color | Frame Grabber Generation | Example |
|---|---|---|---|---|
| Camera Link base configuration | area scan | grayscale | mE4VD4-CL/-PoCL | Section 10.1.1.1, 'Grayscale Camera Link Base Area' |
| Camera Link base configuration | area scan | RGB | mE4VD4-CL/-PoCL | Section 10.1.1.2, 'RGB Camera Link Base Area' |
| Camera Link medium configuration | area scan | grayscale | mE4VD4-CL/-PoCL | Section 10.1.1.3, 'Grayscale Camera Link Medium Area' |
| Camera Link medium configuration | area scan | RGB | mE4VD4-CL/-PoCL | Section 10.1.1.4, 'RGB Camera Link Medium Area' |
| Camera Link full configuration | area scan | grayscale | mE4VD4-CL/-PoCL | Section 10.1.1.5, 'Grayscale Camera Link Full Area' |
| Camera Link base configuration | line scan | grayscale | mE4VD4-CL/-PoCL | Section 10.1.2.1, 'Grayscale Camera Link Base Line' |
| Camera Link base configuration | line scan | RGB | mE4VD4-CL/-PoCL | Section 10.1.2.2, 'RGB Camera Link Base Line' |
| Camera Link medium configuration | line scan | grayscale | mE4VD4-CL/-PoCL | Section 10.1.2.3, 'Grayscale Camera Link Medium Line' |
| Camera Link full configuration | line scan | grayscale | mE4VD4-CL/-PoCL | Section 10.1.2.4, 'Grayscale Camera Link Full Line' |
| GigE Vision | area scan | grayscale | mE4VQ4-GE | Section 10.2.1.1, ' GigE Vision Grayscale Area Scan Cameras ' |
| GigE Vision | area scan | RGB | mE4VQ4-GE | Section 10.2.1.2, ' GigE Vision RGB Area Scan Cameras ' |

| Interface | Sensor | Color | Frame Grabber Generation | Example |
|---|---|---|---|---|
| GigE Vision | line scan | grayscale | mE4VQ4-GE | Section 10.2.2.1, 'GigE Vision Grayscale Line Scan Cameras' |
| GigE Vision | line scan | RGB | mE4VQ4-GE | Section 10.2.2.2, 'GigE Vision RGB Line Scan Cameras' |
| Camera Link base configuration | area scan | grayscale | mE5-MA-VCL/LB-VCL/ mE5VD8-CL/-PoCL | Section 10.3.1.1, 'Grayscale Camera Link Base Area' |
| Camera Link base configuration | area scan | RGB | mE5-MA-VCL/LB-VCL/ mE5VD8-CL/-PoCL | Section 10.3.1.2, 'RGB Camera Link Base Area' |
| Camera Link medium configuration | area scan | grayscale | mE5-MA-VCL/LB-VCL/ mE5VD8-CL/-PoCL | Section 10.3.1.3, 'Grayscale Camera Link Medium Area' |
| Camera Link medium configuration | area scan | RGB | mE5-MA-VCL/LB-VCL/ mE5VD8-CL/-PoCL | Section 10.3.1.4, 'RGB Camera Link Medium Area' |
| Camera Link full configuration | area scan | grayscale | mE5-MA-VCL/LB-VCL/ mE5VD8-CL | Section 10.3.1.5, 'Camera Link Full Area' |
| Camera Link base configuration | line scan | grayscale | mE5-MA-VCL/LB-VCL/ mE5VD8-CL | Section 10.3.2.1, 'Grayscale Camera Link Base Line Scan Cameras ' |
| Camera Link base configuration | line scan | RGB | mE5-MA-VCL/LB-VCL/ mE5VD8-CL | Section 10.3.2.2, 'RGB Camera Link Base Line Scan Cameras ' |
| Camera Link medium configuration | line scan | grayscale | mE5-MA-VCL/LB-VCL/ mE5VD8-CL | Section 10.3.2.3, 'Grayscale Camera Link Medium Line Scan Cameras ' |
| Camera Link medium configuration | line scan | RGB | mE5-MA-VCL/LB-VCL/ mE5VD8-CL | Section 10.3.2.4, 'RGB Camera Link Medium Line Scan Cameras ' |
| Camera Link full configuration | line scan | grayscale | mE5-MA-VCL/LB-VCL/ mE5VD8-CL | Section 10.3.2.5, 'Grayscale Camera Link Full Line Scan Cameras ' |
| CoaXPress | area scan | grayscale/ RGB | mE5-MA-VCX-QP/ mE5VQ8-CXP6D/ iF-CXP12-Q | Section 10.4.1, 'CoaXPress Area Scan Cameras' |
| CoaXPress | line scan | grayscale/ RGB | mE5-MA-VCX-QP/ mE5VQ8-CXP6D/ iF-CXP12-Q | Section 10.4.2, 'CoaXPress Line Scan Cameras' |

Table 10.1. List of Basic Acquisition Examples

# 10.1. Basic Acquisition Examples for Camera Link Cameras for microEnable IV VD4-CL/-PoCL Frame Grabber

You can find Camera Link scan examples for mE4VD4-CL/-PoCL platform in the following sections. In subsection Section 10.1.1, ' Camera Link Area Scan Cameras ' examples for area scan cameras and in subsection Section 10.1.2, 'Camera Link Line Scan Cameras' for line scan cameras are described.

## 10.1.1. Camera Link Area Scan Cameras

The basic acquisition for area scan cameras is very easy. You simply need to select a suitable camera operator. Connect the camera operator to an ImageBuffer and DmaToPC operator. Any processing logic can be placed in between, preferably behind the buffer.

### 10.1.1.1. Grayscale Camera Link Base Area

Simply connect the three operators and parameterize them to meet your requirements. If you are using a bit depth not equal to 8 or 16, you should consider a change of the output bit depth to one of these formats. In tutorial Section 9.1, 'Applet Parameterization' explanations on bit depth modifications can be found.

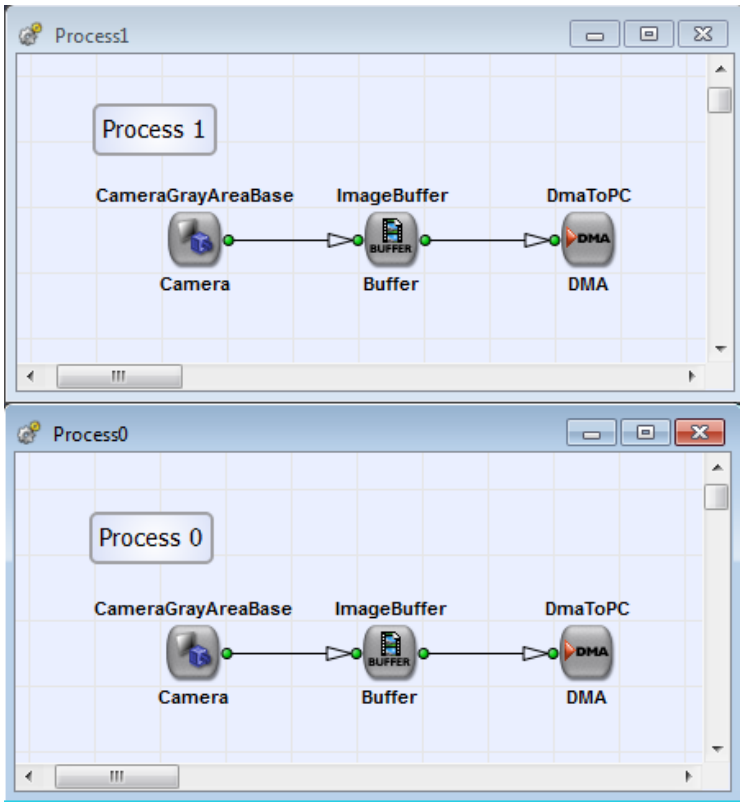



Figure 10.1. Basic Acquisition for Grayscale Camera Link Area Scan Cameras in Base Configuration Mode

The design includes two cameras in two different processes. You can find this example under \examples\Acquisition\BasicAcquisition\mE4VD4-CL\AreaScan\DualBaseAreaGray8.va. Check Section 4.4, 'Multiple Processes' for more information on multiple processes. Under \examples \Acquisition\BasicAcquisition\mE4VD4-CL\AreaScan\BaseAreaGray8.va and \examples\Acquisition \BasicAcquisition\mE4VD4-CL\AreaScan\BaseAreaGray12.va you also have access to two basic single acquisition processes for 8 bit and 12 bit input bit depth.

### 10.1.1.2. RGB Camera Link Base Area

The use of RGB cameras is similar to the use of grayscale cameras. Again, it might be necessary to modify the output bit depth. You can find the following example under \examples\Acquisition \BasicAcquisition\mE4VD4-CL\DualBaseAreaRGB24.va. It is an example with two processes analog to the example in Section 10.1.1.1, 'Grayscale Camera Link Base Area'. An single process example for RGB area scan cameras is placed under \examples\Acquisition\BasicAcquisition\mE4VD4-CL \BaseAreaRGB24.va.

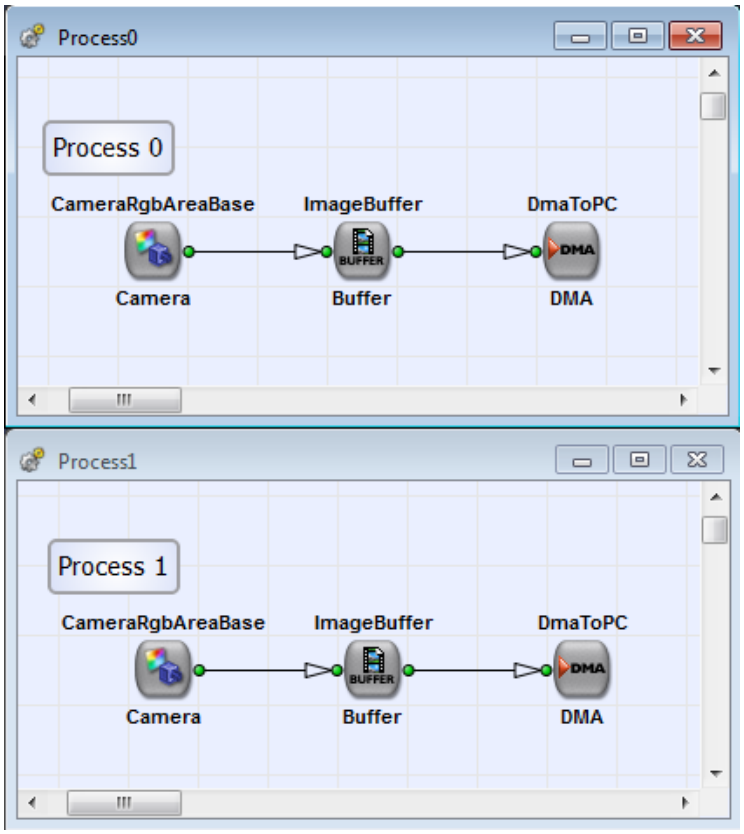



Figure 10.2. Basic Acquisition for RGB Camera Link Area Scan Cameras in Base Configuration Mode

### 10.1.1.3. Grayscale Camera Link Medium Area

The use of medium cameras is similar to the use of cameras in Camera Link base configuration mode. The camera medium configuration camera operator allows a higher bandwidth.

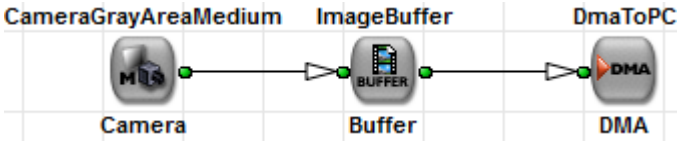



Figure 10.3. Basic Acquisition for Grayscale Camera Link Area Scan Cameras in Medium Configuration Mode

The examples for 8 bit and 12 bit input bit depth are placed under \examples \Acquisition\BasicAcquisition\mE4VD4-CL\AreaScan\MediumAreaGray8.va and \examples\Acquisition \BasicAcquisition\mE4VD4-CL\AreaScan\MediumAreaGray12.va.

### 10.1.1.4. RGB Camera Link Medium Area

Designs for RGB Camera Link cameras in medium configuration mode need a little modification compared to the other modes. For the medium configuration mode up to 85MPixel/s @ 36 bit per pixel i.e. 382.5MB/s can be transfered. The frame grabber RAM operators cannot process this high data rate. Thus, it is not possible to connect a single ImageBuffer operator to the camera module. From the operator documentation of operator *ImageBuffer* we know that only 64 bit can be connected to the buffer input. As we need parallelism 2 and 36 bit, we will get a DRC error. A simple solution to solve the bandwidth bottleneck is the use of two buffers in parallel. One buffer module will store the lower bits (LSBs) while another buffer module is used to store the upper bits (MSBs).

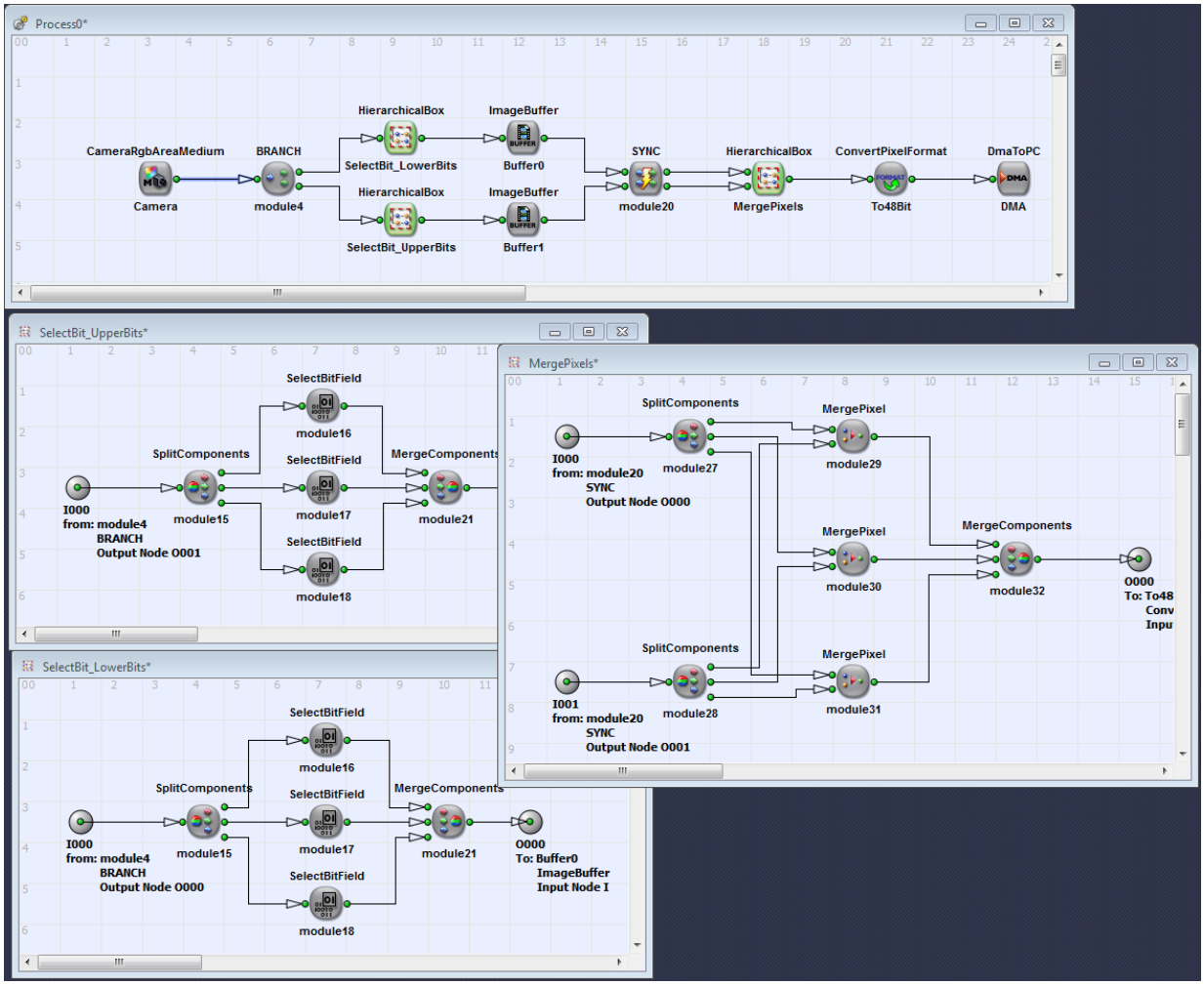



Figure 10.4. Basic Acquisition for RGB Camera Link Area Scan Cameras in Medium Configuration Mode

You can find this example under \examples\Acquisition\BasicAcquisition\mE4VD4-CL\AreaScan \MediumAreaRGB36.va.

### 10.1.1.5. Grayscale Camera Link Full Area

For the microEnable IV frame grabbers, designs for cameras in Camera Link full configuration mode need a little modification compared to cameras in medium or base configuration mode. The full configuration mode represents the fastest Camera Link mode. In theory, up to 850 Mbyte/s can be transfered. The frame grabber RAM operators cannot process this high data rate. Thus, it is not possible to connect a single ImageBuffer operator to the camera module. From the operator documentation of operator *ImageBuffer* we know that only 64 bit can be connected to the buffer input. We will obtain a DRC error when trying to build a design with only one ImageBuffer module. A simple solution to solve

the bandwidth bottleneck is the use of two buffer operators in parallel. One buffer module will store the lower bits (LSBs), while another buffer module is used to store the upper bits (MSBs).

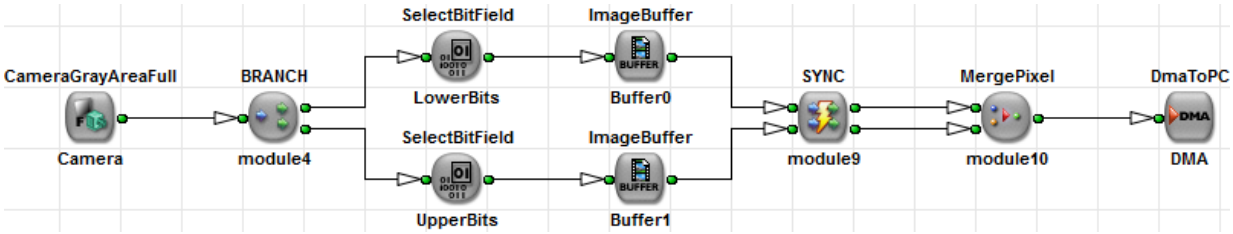



Figure 10.5. Basic Acquisition for Grayscale Camera Link Area Scan Cameras in Full Configuration Mode

The use of a 10 bit Camera Link full camera is similar. Change the properties in the camera module and link to 10 bit and parallelism 12. The *SelectBitField* modules now have to be changed to 5 bit and 5 bit offset. Note that the DMA output is 10 bit in this case. This is a packed format as one byte can contain multiple pixel. If you extend the output to 16 bit you might get a bandwidth limitation.

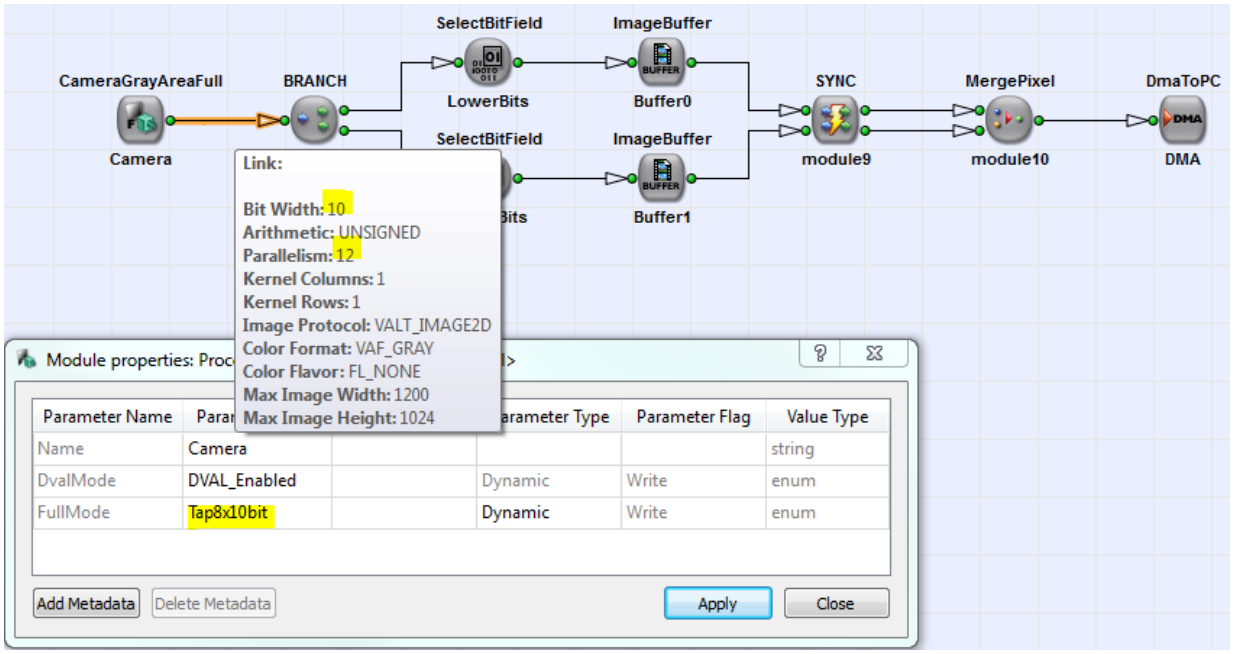



Figure 10.6. Basic Acquisition for Grayscale Camera Link Area Scan Cameras in Full Configuration 10 Bit Mode

You find the two basic acquisition examples for 8 bit and 10 bit input bit depth under \examples\Acquisition\BasicAcquisition\mE4VD4-CL\AreaScan\FullAreaGray8.va and \examples\Acquisition\BasicAcquisition\mE4VD4-CL\AreaScan\FullAreaGray10.va.

## 10.1.2. Camera Link Line Scan Cameras

The acquisition for line scan cameras always requires the cut of the camera lines into images of a specific height. In detail, line scan cameras transfer line by line to the frame grabber. They will not include information on the end or start of a new image. The transfer of data from the frame grabber to the PC is required to be send in packages i.e. frames. Therefore, the lines from line scan cameras have to be assembled into an image of a specific height. There exist numerous possibilities to specify the height. One simple possibility is to accumulate a specific number of lines to form an image, or the image height is determined by other dynamic sources such as external image trigger gate signals. The following Camera Link examples are equipped with a *TrgPortLine* operator (see Section 29.50, 'TrgPortLine'). This operator covers a wide range of trigger functionalities and can be used as an all-

purpose trigger module. Besides the image trigger functionality it can be used to trigger the camera.For more information on the data transfer of line scan cameras Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure' The following designs show simple Camera Link line scan VisualApplets implementations for basic acquisition.

### 10.1.2.1. Grayscale Camera Link Base Line

The example (\examples\Acquisition\BasicAcquisition\mE4VD4-CL\LineScan\DualBaseLineGray8.va ) in the following figure is a dual process design for 8 bit input bit depth. The equivalent single process example is placed under \examples\Acquisition\BasicAcquisition\mE4VD4-CL\LineScan \BaseLineGray8.va.

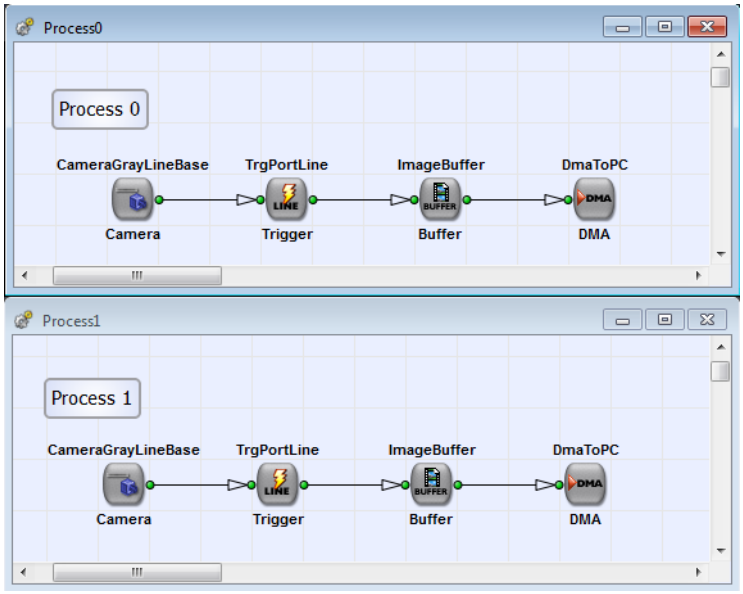



Figure 10.7. Basic Acquisition for Grayscale Camera Link Line Scan Cameras in Base Configuration Mode

### 10.1.2.2. RGB Camera Link Base Line

The following dual process acquisition example for RGB cameras in Camera Link Base configuration mode is equivalent to the grayscale example in Section 10.1.2.1, 'Grayscale Camera Link Base Line'. You can find it under \examples\Acquisition\BasicAcquisition\mE4VD4-CL\LineScan \DualBaseLineRGB24.va.

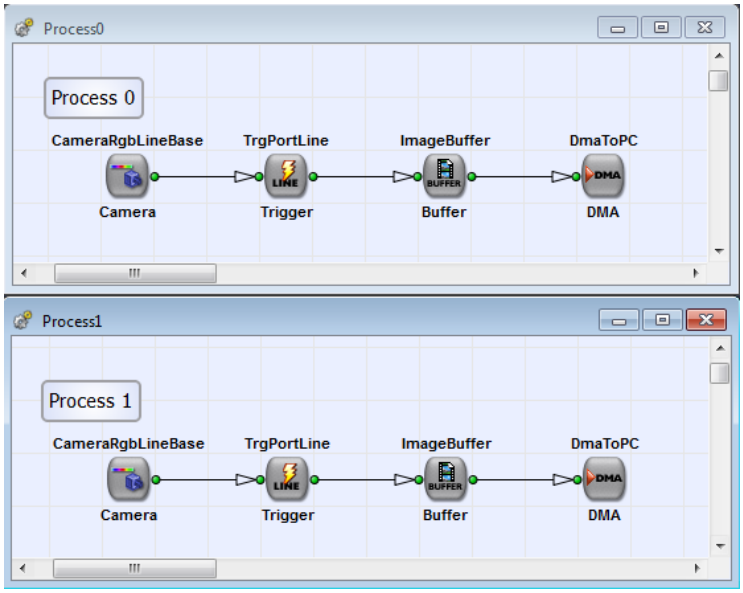



Figure 10.8. Basic Acquisition for RGB Camera Link Line Scan Cameras in Base Configuration Mode

For the single process example see \examples\Acquisition\BasicAcquisition\mE4VD4-CL\LineScan\BaseLineRGB24.va.

### 10.1.2.3. Grayscale Camera Link Medium Line

The following two examples show acquisition designs for 8 bit (\examples\Acquisition\BasicAcquisition\mE4VD4-CL\LineScan\MediumLineGray8.va) and 12 bit (\examples\Acquisition\BasicAcquisition\mE4VD4-CL\LineScan\MediumLineGray12.va) input bit depth.

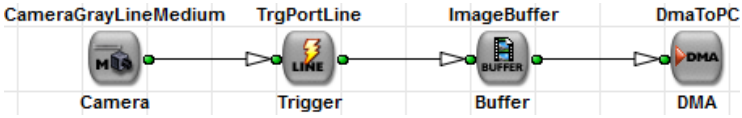



Figure 10.9. Basic Acquisition for Grayscale Camera Link Line Scan Cameras in Medium Configuration Mode

For medium 12 bit cameras two buffers are required as shown in the example for the area scan cameras (see Section 10.1.1.4, 'RGB Camera Link Medium Area').

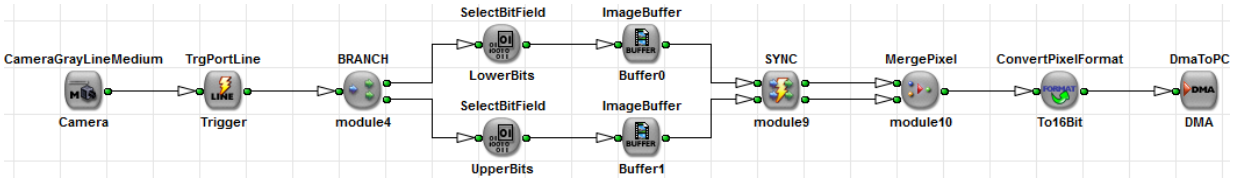



Figure 10.10. Basic Acquisition for Grayscale 12 Bit Camera Link Line Scan Cameras in Medium Configuration Mode

### 10.1.2.4. Grayscale Camera Link Full Line

The example in the following figure presents an acquisition design in the Camera Link full configuration mode for 8 bit (\examples\Acquisition\BasicAcquisition\mE4VD4-CL\LineScan\FullLineGray8.va) and 10 bit (\examples\Acquisition\BasicAcquisition\mE4VD4-CL\LineScan\FullLineGray8.va) input bit width.

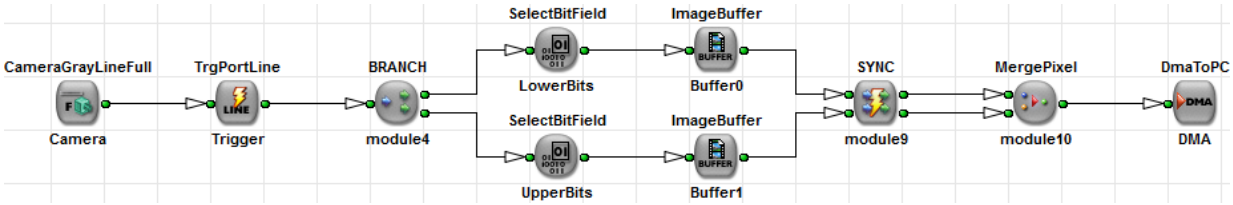



Figure 10.11. Basic Acquisition for Grayscale Camera Link Line Scan Cameras in Full Configuration Mode

## 10.2. Basic Acquisition Examples for GigE Vision Cameras for microEnable IV Frame Grabber

You can find GigE Vision scan examples for mE4VQ4-GE platforms in the following sections. In Section 10.2.1, ' GigE Vision Area Scan Cameras ' or Section 10.2.2, 'GigE Vision Line Scan Cameras' examples for area scan cameras or line scan cameras are described.

### 10.2.1. GigE Vision Area Scan Cameras

The designs for acquisition of GigE Vision cameras is very simple as can be seen in the following figures. Note that the microEnable IV VQ4-GE/-GPoE allows the connection of up to four cameras with gigabit ethernet interface. Thus, user can use all four cameras at the same time in either four different processes individually or by combining their images and use a common processing. Check Section 4.4, 'Multiple Processes' for more information on multiple processes and Section 9.3.1, ' Synchronizing Cameras ' for more information on how to combine the images of multiple cameras.

#### 10.2.1.1. GigE Vision Grayscale Area Scan Cameras

The example "DualAreaGray8.va" is a two process acquisition design for two grayscale GigE cameras with 8 bit pixel depth. The designs "QuadAreaGray8.va" and "QuadAreaGray12.va" are two 4 process designs for 4 GigE area scan cameras with 8 bit and 12 bit grayscale values. You can find the examples under \examples\Acquisition\BasicAcquisition\mE4VQ4-GE\Area. In Fig. 10.12 you can se the basic design structure of the 4 processes design "QuadAreaGray8.va".

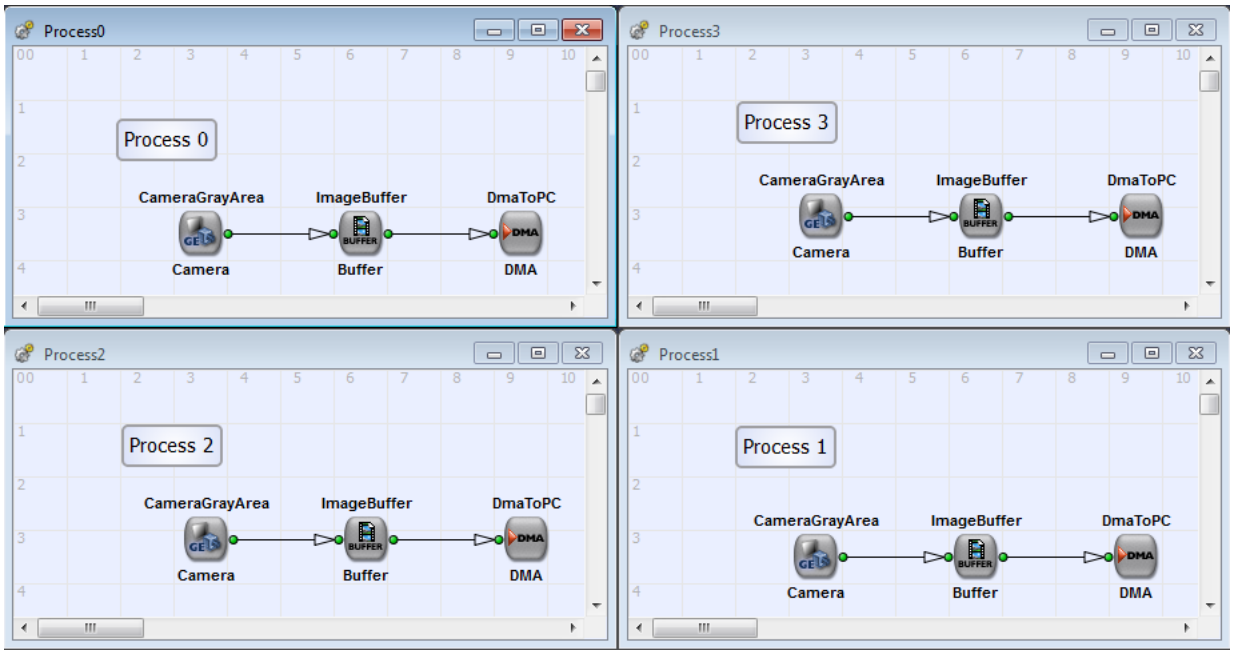



Figure 10.12. Basic Acquisition for Grayscale GigE Vision Area Scan Cameras

### 10.2.1.2. GigE Vision RGB Area Scan Cameras

The examples "DualAreaRGB24.va" and "DualAreaRGB36.va" are dual process designs for RGB image acquisition for 24 bit (8 bit per color component) and 36 bit (12 bit per color component) pixel depth. The design "QuadAreaRGB24.va" is a quad process design and equivalent to the example in Section 10.2.1.1, ' GigE Vision Grayscale Area Scan Cameras '. You can see its basic design structure in Fig. 10.13. Please find the examples under \examples\Acquisition\BasicAcquisition\mE4VQ4-GE\Area.

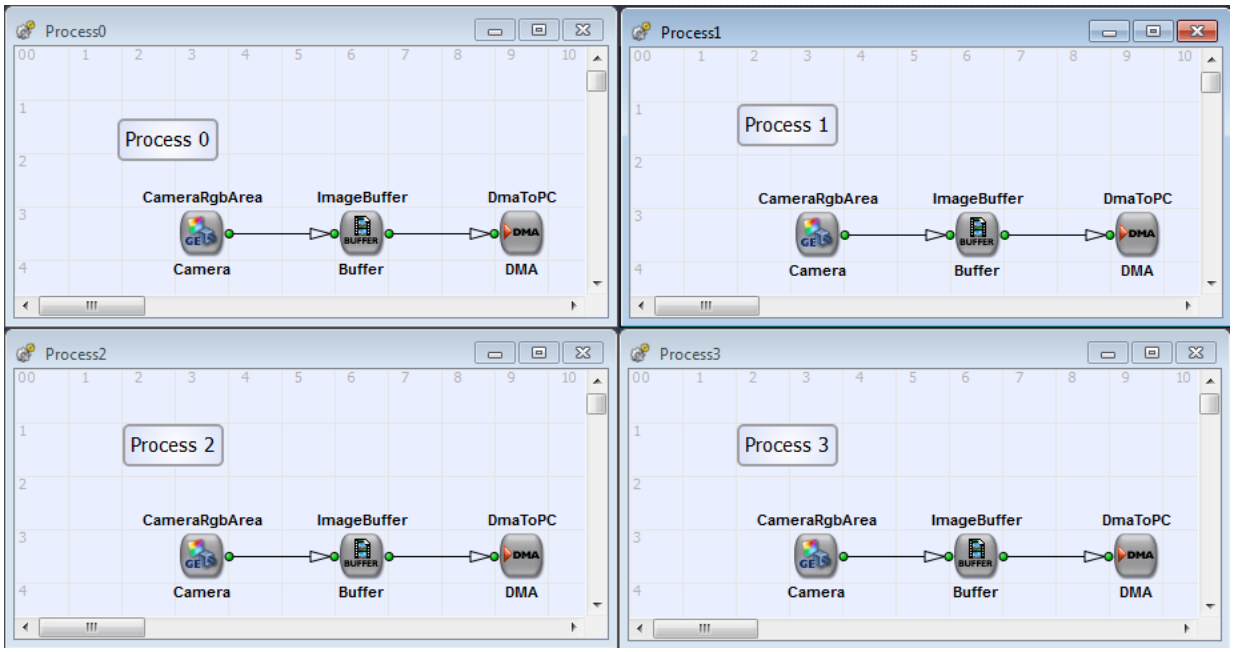



Figure 10.13. Basic Acquisition for RGB GigE Vision Area Scan Cameras

## 10.2.2. GigE Vision Line Scan Cameras

The acquisition for line scan cameras always requires the cut of the camera lines into images of a specific height. In detail, line scan cameras transfer line by line to the frame grabber. They will not include information on the end or start of a new image. The transfer of data from the frame grabber to the PC is required to be send in packages i.e. frames. Therefore, the lines from line scan cameras have to be assembled into an image of a specific height. The GigE Vision examples use the *SplitImage* operator (See Section 32.24, 'SplitImage') to set a parameter defined image height to all acquired images. Another possibility is the usage of operator *SignalGate* (See Section 31.25, 'SignalGate'). For more information on the data transfer of line scan cameras Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure' The following designs show simple Camera Link line scan VisualApplets implementations for basic acquisition. The following designs show the basic acquisition for GigE Vision line scan cameras.

### 10.2.2.1. GigE Vision Grayscale Line Scan Cameras

This example is a four processes design for a grayscale camera with 8 bit. You can find it under \examples\Acquisition\BasicAcquisition\mE4VQ4-GE\QuadLineGray8.va.

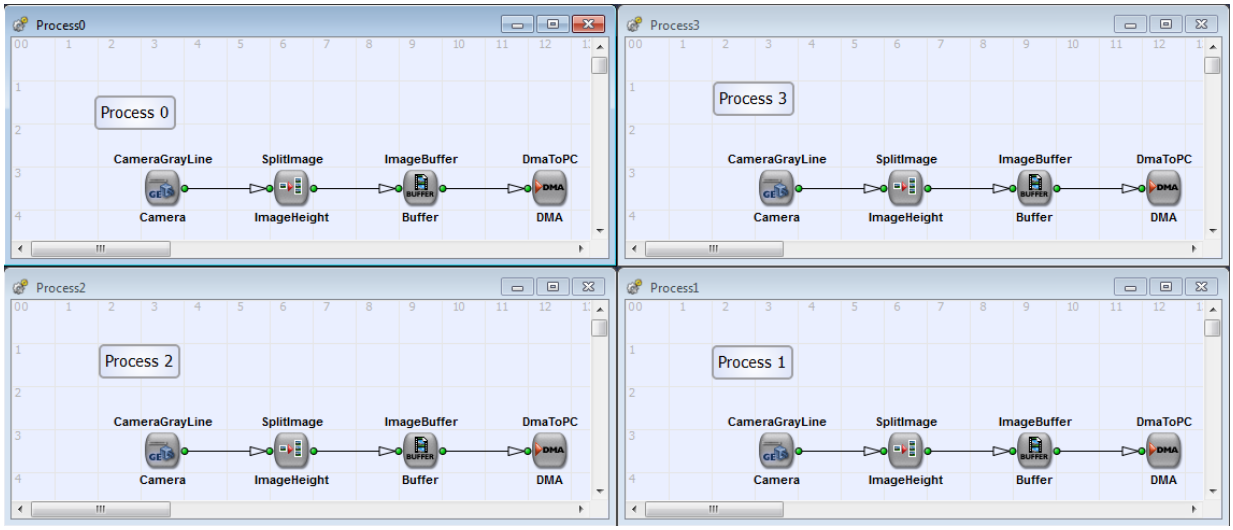



Figure 10.14. Basic Acquisition for Grayscale GigE Vision Line Scan Cameras

### 10.2.2.2. GigE Vision RGB Line Scan Cameras

This example is a four process design for a RGB camera . You can find it under \examples\Acquisition \BasicAcquisition\mE4VQ4-GE\QuadLineRGB.va.

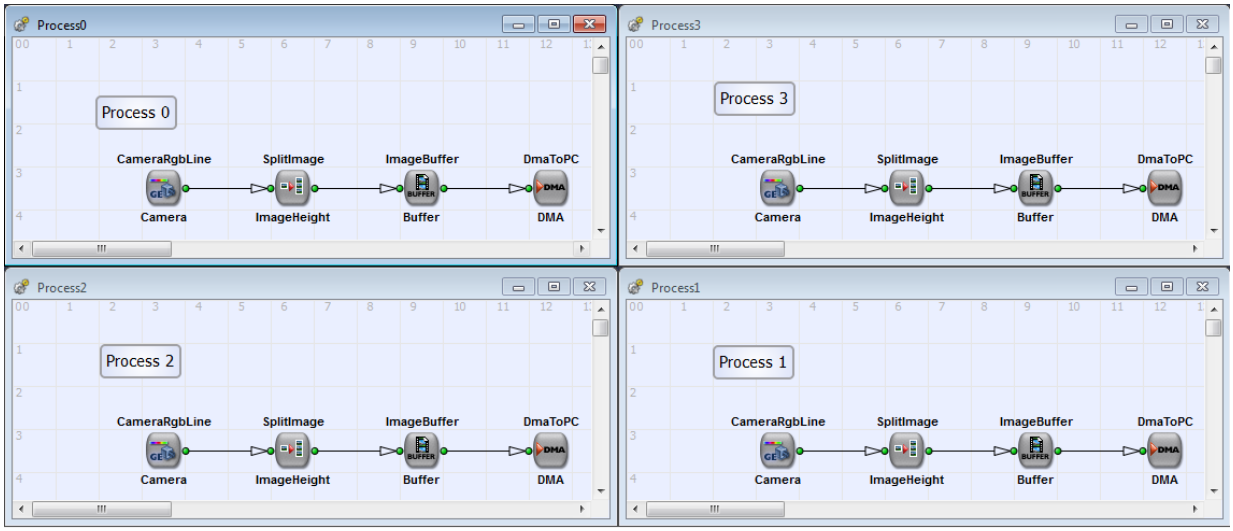



Figure 10.15. Basic Acquisition for RGB GigE Vision Line Scan Cameras

## 10.3. Basic Acquisition Examples for Camera Link Cameras for marathon, LightBridge and ironman Frame Grabbers

You can find Camera Link scan examples for marathon (mE5-MA-VCL), LightBridge (LB-VCL) and ironman (mE5VD8-CL/-PoCL) platforms in the following sections. The basic acquisition examples are very similar to the ones on the mE4VD4-CL/-PoCL platform. In 10.3.1 and in 10.3.2 examples for area and line scan cameras are described.

### 10.3.1. Camera Link Area Scan Cameras

The basic acquisition for area scan cameras is very easy. You simply need to select a suitable camera operator. Connect the camera operator to an ImageBuffer and DmaToPC operator. Any processing logic can be placed in between, preferably behind the buffer. In the following basic examples for Camera Link configuration mode base, medium and full are presented.

#### 10.3.1.1. Grayscale Camera Link Base Area

Simply connect the three operators and parameterize them to meet your requirements. If you are using a bit depth no equal to eight or 16, you should consider a change of the output bit depth to one of these formats. In tutorial Section 9.1, 'Applet Parameterization' explanations on bit depth modifications can be found.

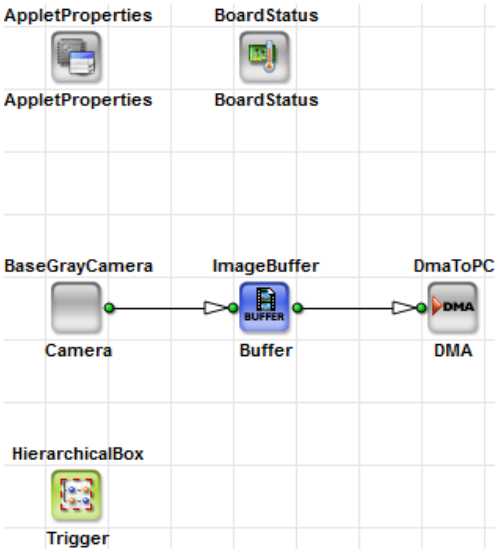



Figure 10.16. Basic Acquisition for Grayscale Camera Link Area Scan Cameras in Base Configuration Mode on LightBridge VCL, marathon VCL and ironman VCL

You can find the examples "BaseAreaGray8.va", "DualBaseAreaGray8.va" and "BaseAreaGray12.va" for 8 bit and 12 bit under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Area and \examples\Acquisition\BasicAcquisition\mE5VD8-CL\Area. The example "DualBaseAreaGray8.va" is a dual process design. Please read for information purpose under Appendix A, '*Device Resources*' the concept of shared memory on the microEnable 5 marathon and LightBridge platforms. For more information about the Trigger box in "BaseAreaGray8.va" and how to trigger Camera Link area scan cameras see Section 12.15, 'Trigger'.

#### 10.3.1.2. RGB Camera Link Base Area

Please find the examples "BaseAreaRGB24.va" and "DualBaseAreaRGB24.va" for RGB 24 bit for acquisition with a camera in Camera Link base configuration under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Area and \examples\Acquisition\BasicAcquisition\mE5VD8-CL\Area. The example "DualBaseAreaRGB24.va" is a dual process design. Please read for information purpose under Appendix A, '*Device Resources*' the concept of shared memory on the microEnable 5 marathon and LightBridge platforms. For more information on how to trigger your Camera Link camera see Section 12.15, 'Trigger'.

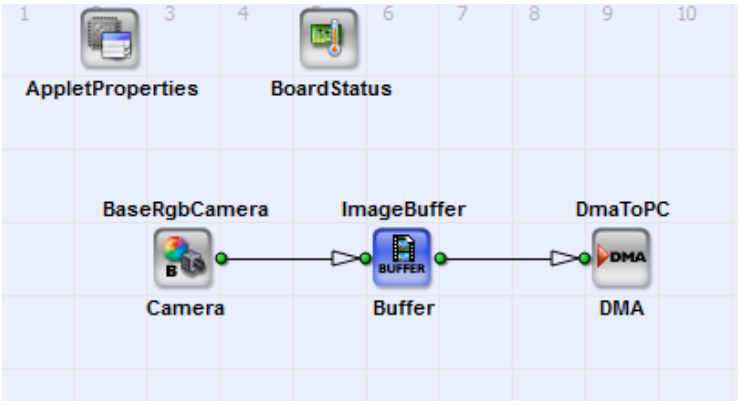



Figure 10.17. Basic Acquisition for RGB Camera Link Area Scan Cameras in Base Configuration Mode on LightBridge VCL, marathon VCL and ironman VCL

### 10.3.1.3. Grayscale Camera Link Medium Area

The use of medium cameras is similar to the use of cameras in Camera Link base configuration mode. The camera medium configuration camera operator allows a higher bandwidth. You find the example designs "MediumAreaGray8.va" and "MediumAreaGray12.va" for 8 bit and 12 bit for the medium configuration of a Camera Link grayscale camera under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Area and \examples\Acquisition\BasicAcquisition\mE5VD8-CL\Area.

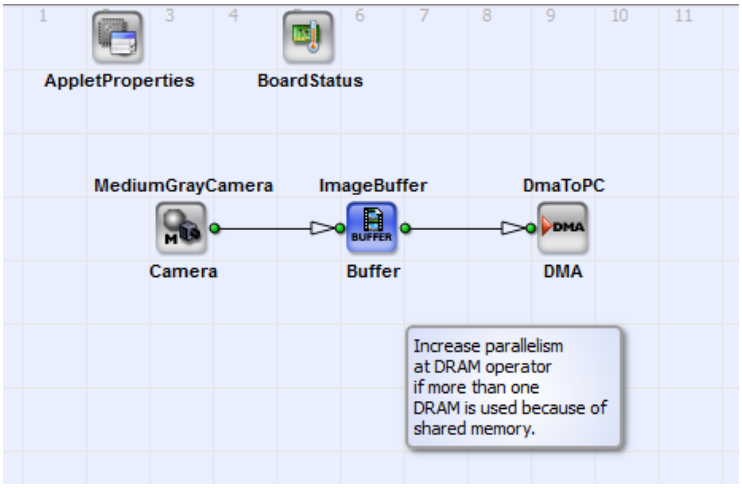



Figure 10.18. Basic Acquisition for Grayscale Camera Link Area Scan Cameras in Medium Configuration Mode on LightBridge VCL, marathon VCL and ironman VCL

### 10.3.1.4. RGB Camera Link Medium Area

Please find the example "MediumAreaRGB36.va" for 36 bit input bit depth for acquisition with a camera in Camera Link medium configuration under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Area and \examples\Acquisition\BasicAcquisition\mE5VD8-CL\Area. In comparison to the medium RGB design for the mE IV VD4 CL/-Pocl (see section 10.1.1.4) the design requires only one DRAM element at a parallelism of 2 and 36 bit pixel depth due to the allowed DRAM data width of 256 bit (mE5-MA-VCL) and 128 bit on (mE5VD8-CL/-PoCL) (see Appendix A, '*Device Resources*').

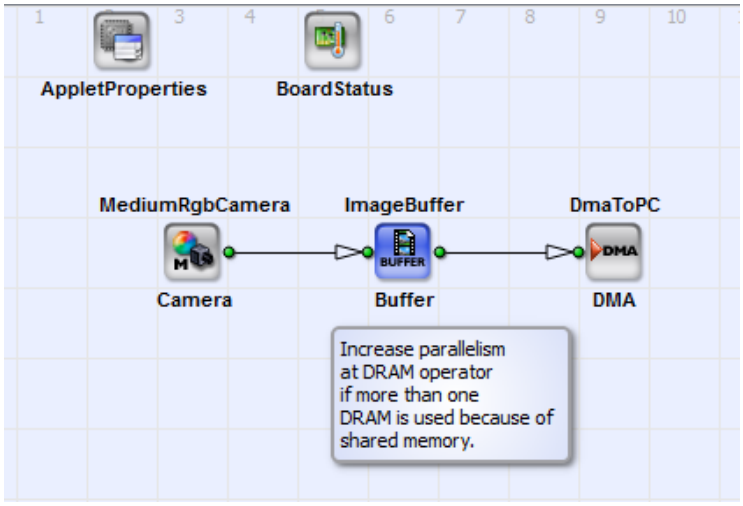



Figure 10.19. Basic Acquisition for RGB Camera Link Area Scan Cameras in Base Configuration Mode on LightBridge VCL, marathon VCL and ironman VCL

### 10.3.1.5. Camera Link Full Area

For the marathon, LightBrige and ironman frame grabber the usage of an area scan camera in full configuration mode is simple. The following figure shows the usage. Ensure to set the VALT_IMAGE2D image protocol in the link properties. You can find the examples "FullAreaGray8.va" and "FullAreaGray10.va" for 8 and 10 bit pixel depth under \examples\Acquisition\BasicAcquisition \mE5-MA-VCL\Area and \examples\Acquisition\BasicAcquisition\mE5VD8-CL\Area.

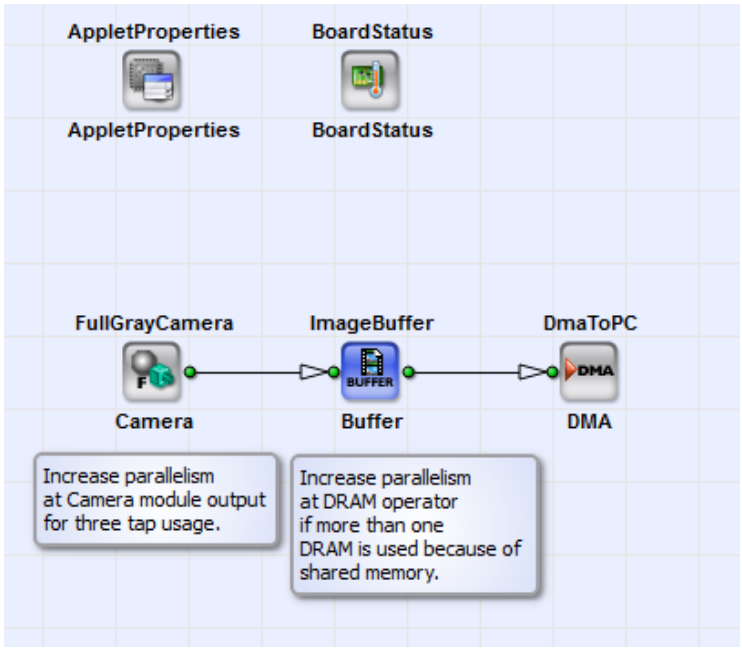



Figure 10.20. Basic Acquisition Design for marathon VCL, LightBridge VCL and ironman VCL Frame Grabber for Camera Link Area Scan Cameras in Full Configuration Mode

## 10.3.2. Camera Link Line Scan Cameras

The acquisition for line scan cameras always requires the cut of the camera lines into images of a specific height. In detail, line scan cameras transfer line by line to the frame grabber. The transfer of data from the frame grabber to the PC is required to be send in packages i.e. frames. Therefore,

the lines from line scan cameras have to be assembled into an image of a specific height. There exist numerous possibilities to specify the height. One simple possibility is to accumulate a specific number of lines to form an image, or the image height is determined by other dynamic sources such as external image trigger gate signals. For the microEnable 5, the TrgPortLine operator (see Section 10.1.2, 'Camera Link Line Scan Cameras') is not available. The following example converts the line data from camera to 2D image data in *SplitImage*. Ensure to set the image protocol output of the camera operator to VALT_LINE1D. The examples are for grayscale cameras for Camera Link base and full configuration mode. Please adapt the example design for RGB cameras and Camera Link medium configuration equivalently.

### 10.3.2.1. Grayscale Camera Link Base Line Scan Cameras

The examples "BaseLineGray8.va", "DualBaseLineGray8.va" and "BaseLineGray12.va" for 8 bit and 12 bit pixel depth are basic acquisition designs for greyscale line cameras in Camera Link base configuration mode. The design "DualBaseLineGray8.va" is a dual process design. Please read for information purpose under Appendix A, '*Device Resources*' the concept of shared memory on the microEnable 5 marathon and LightBridge platforms. You can find the examples under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Line and \examples\Acquisition \BasicAcquisition\mE5VD8-CL\Line.

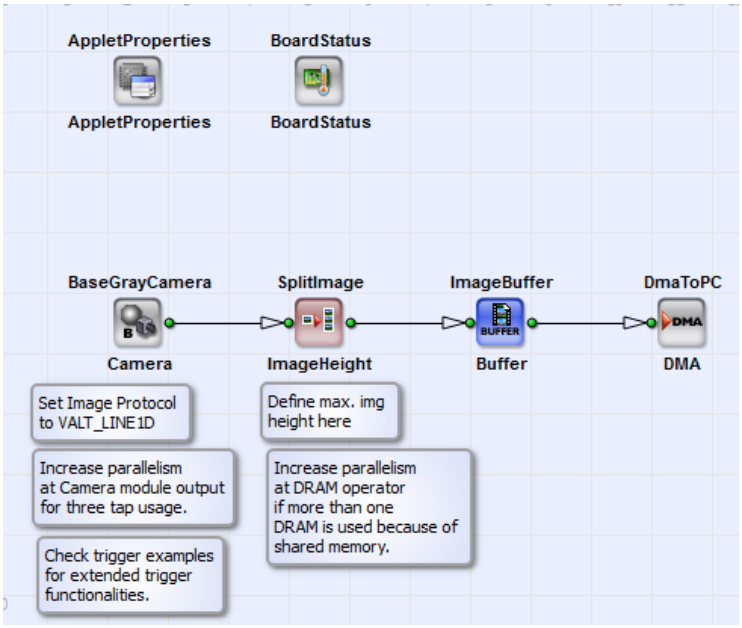



Figure 10.21. Basic Acquisition Design for marathon, LightBridge and ironman Frame Grabber for Grayscale Camera Link Line Scan Cameras in Base Configuration Mode

For the LightBridge, marathon and ironman frame grabbers we use as an alternative *TrgBoxLine* instead of TrgPortLine. See the following figure. Ensure to set the image protocol output of the camera operator to VALT_LINE1D.

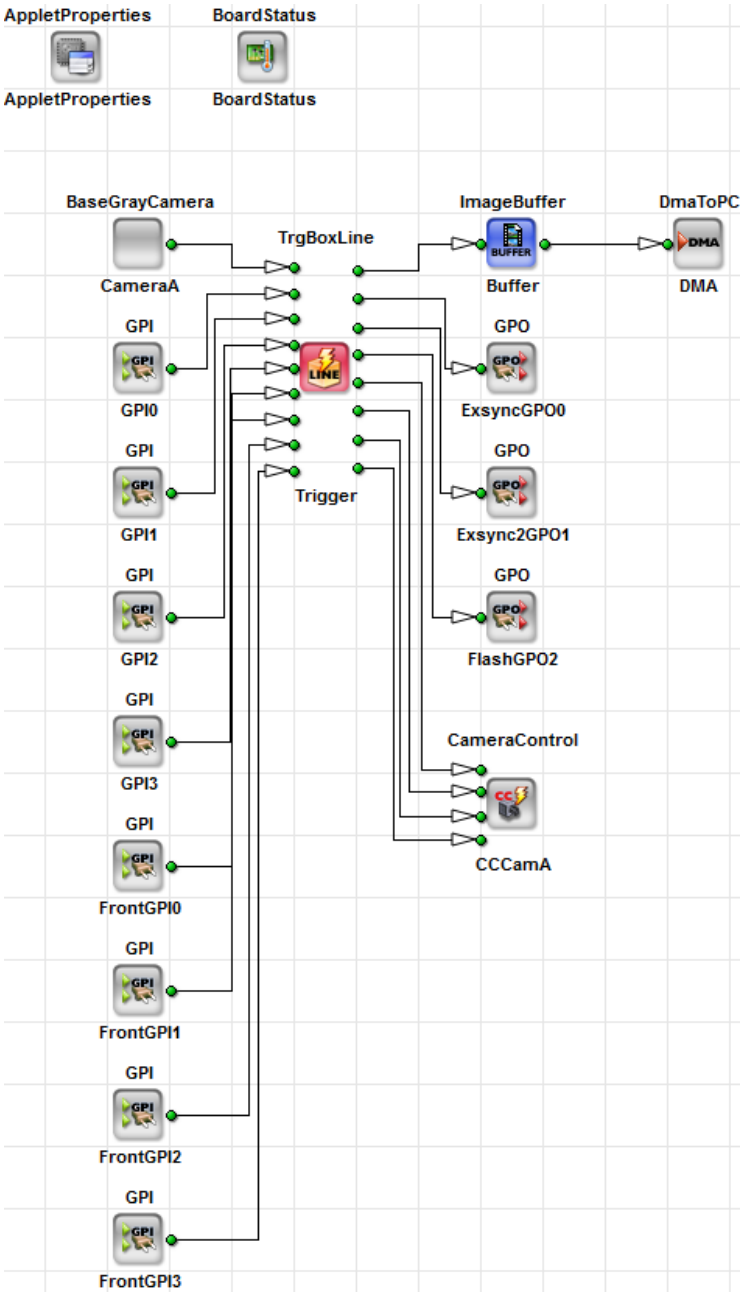



Figure 10.22. Basic Acquisition for Grayscale Camera Link Line Scan Cameras in Base Configuration Mode on the LightBridge VCL, marathon VCL and ironman VCL

## 10.3.2.2. RGB Camera Link Base Line Scan Cameras

You find the examples "BaseLineRGB24.va" and "DualBaseLineRGB24.va" for 24 bit pixel depth (8 bits per color component) for acquisition with RGB line cameras in Camera Link base configuration mode under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Line and \examples\Acquisition \BasicAcquisition\mE5VD8-CL\Line. They are equivalent to the grayscale line scan acquisition design (see section 10.3.2.1). Please read in section Appendix A, '*Device Resources*' the concept of shared memory on the microEnable 5 marathon and LightBridge platforms. This information is highly relevant for multiple process designs (e.g. "DualBaseLineRGB24.va") or designs which use multiple DRAM elements.

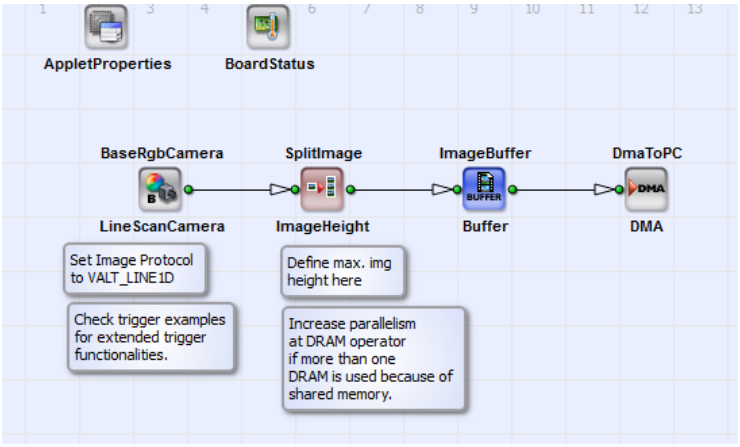



Figure 10.23. Basic Acquisition Design for marathon, LightBridge and ironman Frame Grabber for RGB Camera Link Line Scan Cameras in Base Configuration Mode

### 10.3.2.3. Grayscale Camera Link Medium Line Scan Cameras

The examples "MediumLineGray8.va" and "MediumLineGray12.va" for 8 bit and 12 bit pixel depth are basic acquisition designs for greyscale line cameras in Camera Link medium configuration mode. They are equivalent to the designs for base configuration but allow a higher data rate. You can find the examples under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Line and \examples\Acquisition \BasicAcquisition\mE5VD8-CL\Line.

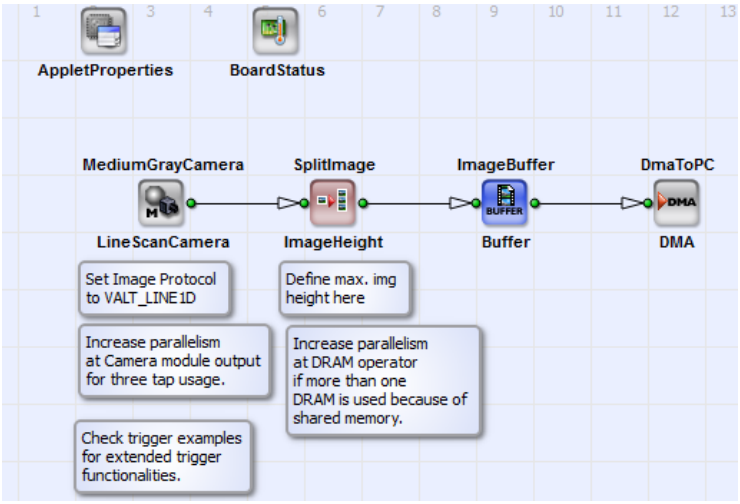



Figure 10.24. Basic Acquisition Design for marathon, LightBridge and ironman Frame Grabber for Grayscale Camera Link Line Scan Cameras in Base Configuration Mode

### 10.3.2.4. RGB Camera Link Medium Line Scan Cameras

You find the example "MediumLineRGB36.va" for 36 bit pixel depth (12 bits per color component) for acquisition with RGB line scan cameras in Camera Link medium configuration mode under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Line and \examples\Acquisition \BasicAcquisition\mE5VD8-CL\Line. It is equivalent to the grayscale line scan acquisition designs (see section 10.3.2.3).

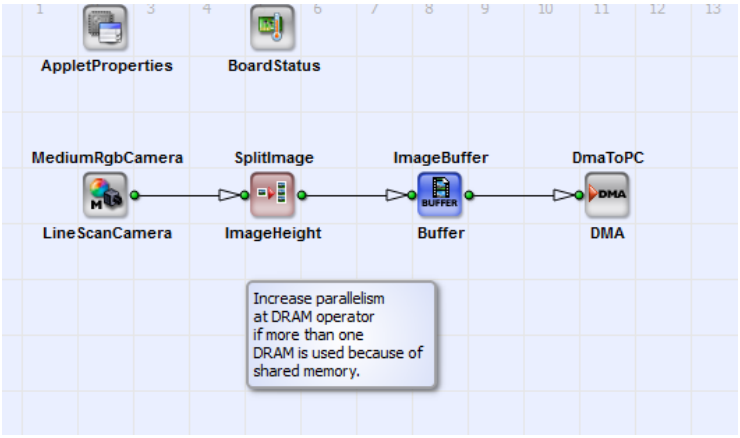



Figure 10.25. Basic Acquisition Design for marathon, LightBridge and ironman Frame Grabber for RGB Camera Link Line Scan Cameras in Base Configuration Mode

### 10.3.2.5. Grayscale Camera Link Full Line Scan Cameras

You can find the Camera Link full line scan acquisition design examples "FullLineGray8.va" and "FullLineGray10.va" for 8 bit and 10 bit pixel depth under \examples\Acquisition\BasicAcquisition\mE5-MA-VCL\Line\FullLineGray8.va \examples\Acquisition\BasicAcquisition\mE5VD8-CL\Line. If you want to perform RGB image acquisition instead of grayscale, simply replace the operator "FullGrayCamera" by the operator "FullRgbCamera".

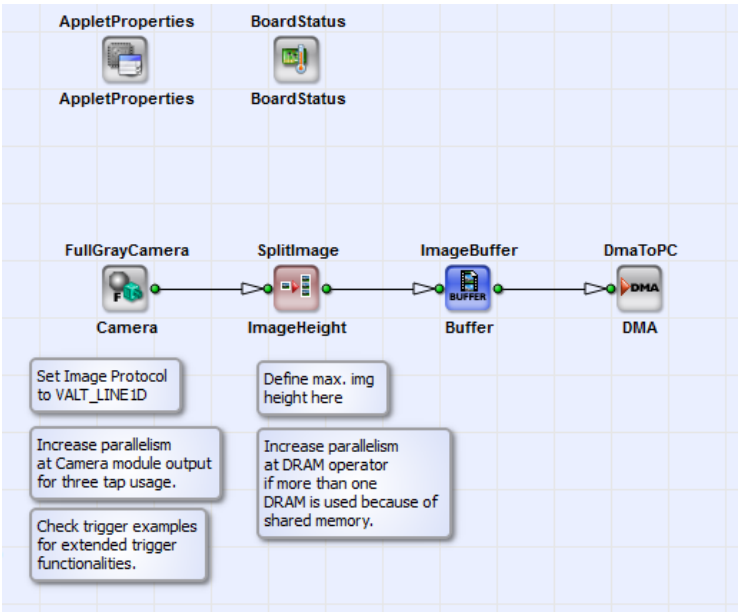



Figure 10.26. Basic Acquisition for marathon, LightBridge and ironman Frame Grabber for Camera Link Line Scan Cameras in Full Configuration Mode

## 10.4. Basic Acquisition Examples for CoaXPress Cameras for marathon and ironman Frame Grabbers

In the following you can find basic acquisition examples for CoaXPress cameras for the marathon and ironman frame grabbers mE5-MA-VCX-QP and mE5VQ8-CXP6D. In Section 10.4.1, 'CoaXPress Area Scan Cameras' designs for area scan cameras and in Section 10.4.2, 'CoaXPress Line Scan Cameras' designs for line scan cameras are presented.

### 10.4.1. CoaXPress Area Scan Cameras

Basic acquisition for CoaXPress cameras on the marathon mE5-MA-VCX-QP and ironman mE5VQ8-CXP6D platforms is very simple. You have to use either operator *CXPSingleCamera*, *CXPDualCamera* or *CXPQuadCamera*. With "double-mouse-click" on the camera operator you can choose the format type (i.e. Gray, RGB, or also Bayer Raw) and the format mode i.e the corresponding bit depth. Therefore for the CoaXPress cameras no separate camera operators for grayscale and RGB acquisition exist. In the following sections example designs for image acquisition with the three camera types (CXPSingleCamera, CXPDualCamera and CXPQuadCamera) are introduced. Keep in mind to set the correct link format in the output links of the camera operators. Ensure that the maximum image width and the correct image protocol is set. Some operators output their data at a parallelism which is not a multiple of a power of two value. As it is more convenient to work on power of two parallelism, the parallelism can be increase with a *PARALLELup* operator. The successive *ImageBuffer* will then cut the correct ROI and remove dummy pixel which might have been added due to parallelism conversions.

#### 10.4.1.1. Basic Acquisition Example for Single Line CoaXPress Area Scan Cameras

Please find the basic acquisition designs "SingleCXP6x1AreaGray8", "SingleCXP6x1AreaGray10.va", "SingleCXP6x1AreaGray12.va" (grayscale 8 bit, 10 bit and 12 bit) and "SingleCXP6x1AreaRGB24.va" (color 24 bit) for one single link aggregation camera for the mE5VQ8-CXP6D and mE5-MA-VCX-QP platforms under \examples\Acquisition\BasicAcquisition\mE5VQ8-CXP6D\Area and \examples\Acquisition\BasicAcquisition\mE5-MA-VCX-QP\Area. The designs for the two platforms are equivalent. The following figure shows the basic design structure of the example designs. The examples for four single link aggregation cameras are the designs "QuadCXP6x1AreaGray12.va" (grayscale 12 bit) and "QuadCXP6x1AreaRGB36.va" (color 36 bit with 12 bit per component) and are located in the same folder. Please read the shared memory concept for the mE5-MA-VCX-QP platform under Appendix A, '*Device Resources*'. This information is highly relevant when you use multiple DRAM elements in one design.

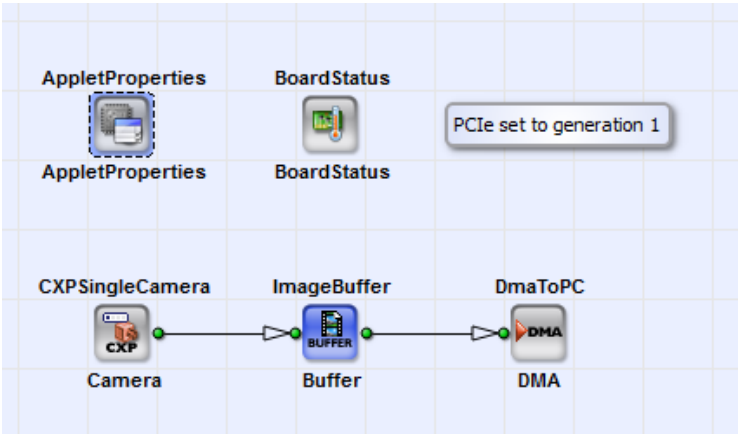



Figure 10.27. Basic Acquisition for Grayscale CoaxPress Area Scan Cameras in 6GBit/s Mode with Link Aggregation 1 on the ironman Frame Grabber

#### 10.4.1.2. Basic Acquisition Examples for two Dual Line CoaXPress Area Scan Cameras

Please find the two example designs "DualCXP6x2AreaGray8.va" (grayscale 8 bit) and "DualCXP6x2AreaRGB24.va" (RGB 24 bit, 8 bits per color component) for the mE5VQ8-CXP6D and mE5-MA-VCX-QP platform under \examples\Acquisition\BasicAcquisition\mE5VQ8-CXP6D\Area and \examples\Acquisition\BasicAcquisition\mE5-MA-VCX-QP\Area. The designs are dual process designs for two CoaXPress cameras with dual link aggregation. Please read also under Appendix A, '*Device Resources*' information on the shared memory concept for the mE5-MA-VCX-QP platform. The following figure shows the basic acquisition design for the design "DualCXP6x2AreaGray8.va".

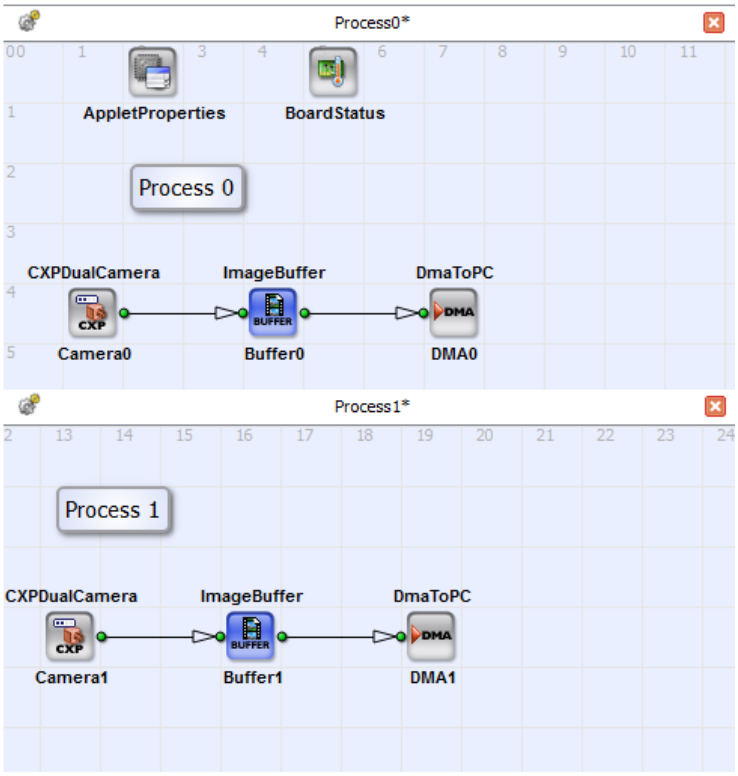



Figure 10.28. Basic Acquisition for RGB CoaxPress Area Scan Cameras in 6 Gbit/s Mode with Link Aggregation 2 on the ironman Frame Grabber

### 10.4.1.3. Basic Acquisition Examples for One Quad Line CoaXPress Area Scan Camera

"SingleCXP6x4AreaGray8.va" (grayscale 8 bit) and "SingleCXP6x4AreaRGB36.va" (RGB with 12 bits per color component) are example designs for one camera with 4 link aggregation. You can find the designs for the mE5VQ8-CXP6D and mE5-MA-VCX-QP platform under \examples\Acquisition \BasicAcquisition\mE5VQ8-CXP6D\Area and \examples\Acquisition\BasicAcquisition\mE5-MA-VCX-QP \Area. The designs for the two platforms differ but follow the same principle. In Fig. 10.29 you can see the basic design structure of the design "SingleCXP6x4AreaGray8.va" for the mE5VQ8-CXP6D platform. As the memory bandwidth (128 bit RAM data width at 3.2 GB/s) for the 6 Gbit/s with link aggregation 4 is not sufficient with a single buffer, two buffers have to be used in parallel for the mE5VQ8-CXP6D platform. For the mE5-MA-VCX-QP platform only one DRAM element is necessary to achieve the required bandwidth of 6Gbit/s per line. Here up to 512 bit RAM data width at a RAM bandwidth of 12.8 GB/s are possible. Please read for more information on the devices resources Appendix A, '*Device Resources*'. In Fig. 10.30 you can see the basic design structure of the example "SingleCXP6x4AreaGray8.va" on the mE5-MA-VCX-QP platform.

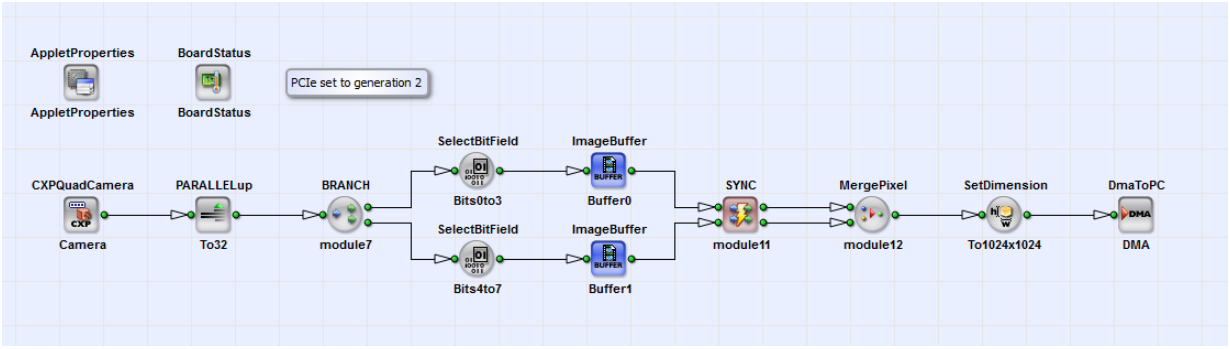



Figure 10.29. Basic Acquisition for Grayscale CoaxPress Area Scan Cameras in 6 Gbit/s Mode with Link Aggregation 4 on the ironman Frame Grabber

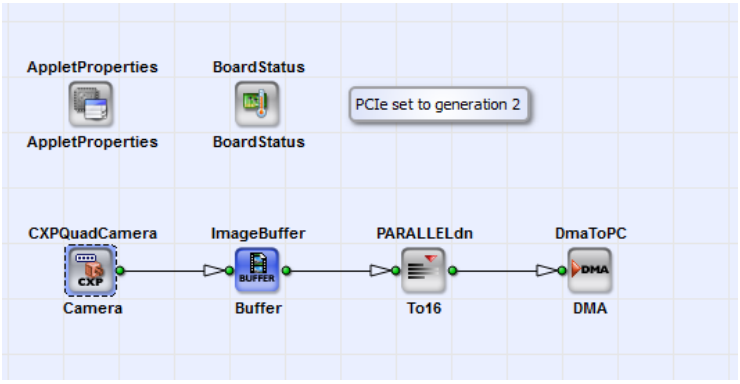



Figure 10.30. Basic Acquisition for Grayscale CoaxPress Area Scan Cameras in 6 Gbit/s Mode with Link Aggregation 4 on the marathon Frame Grabber

## 10.4.2. CoaXPress Line Scan Cameras

The examples presented in the following subsections are basic acquisition designs for grayscale line scan CoaXPress cameras. They are equivalent to the designs in section Section 10.4.1, 'CoaXPress Area Scan Cameras'. Just set the camera operator to VALT_LINE1D and add the operator "SplitImage".

### 10.4.2.1. Basic Acquisition Example for Single Line CoaXPress Line Scan Cameras

Please find the basic acquisition designs "SingleCXP6x1LineGray8", "SingleCXP6x1LineGray10.va", "SingleCXP6x1LineGray12.va" (grayscale 8 bit, 10 bit and 12 bit) and "SingleCXP6x1LineRGB24.va" (color 24 bit) for one single link aggregation camera for the mE5VQ8-CXP6D and mE5-MA-VCX-QP platforms under \examples\Acquisition\BasicAcquisition\mE5VQ8-CXP6D\Line and \examples\Acquisition\BasicAcquisition\mE5-MA-VCX-QP\Line. The designs for the two platforms are equivalent. The following figure shows the basic design structure of the example designs. The examples for four single link aggregation cameras are the designs "QuadCXP6x1LineGray12.va" (grayscale 12 bit) and "QuadCXP6x1LineRGB36.va" (color 36 bit) and are located in the same folder. Please read the shared memory concept for the mE5-MA-VCX-QP platform under Appendix A, '*Device Resources*'. This information is highly relevant when you use multiple DRAM elements in one design.

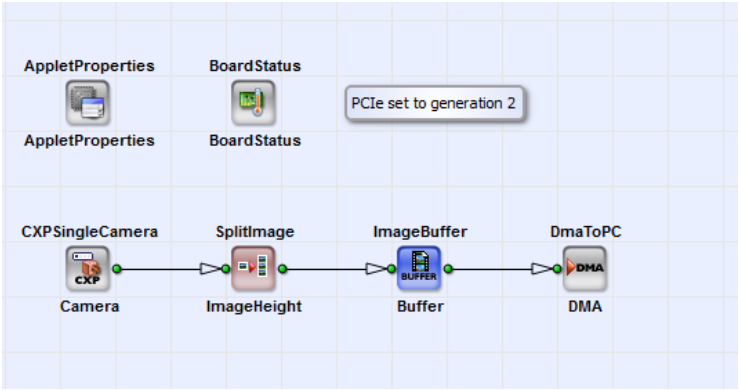



Figure 10.31. Basic Acquisition for Grayscale CoaxPress Line Scan Cameras in 6 GBit/s Mode with Link Aggregation 1 on the ironman Frame Grabber

### 10.4.2.2. Basic Acquisition Examples for two Dual Line CoaXPress Line Scan Cameras

Please find the two example designs "DualCXP6x2LineGray8.va" (grayscale 8 bit) and "DualCXP6x2LineRGB24.va" (RGB 24 bit, 8 bits per color component) for the mE5VQ8-CXP6D and mE5-MA-VCX-QP platform under \examples\Acquisition\BasicAcquisition\mE5VQ8-CXP6D\Line and \examples\Acquisition\BasicAcquisition\mE5-MA-VCX-QP\Line. The designs are dual process designs for two CoaXPress cameras with dual link aggregation. Please read also under Appendix A, '*Device Resources*' information on the shared memory concept for the mE5-MA-VCX-QP platform. Fig. 10.32 shows the basic acquisition design for the design "DualCXP6x2LineGray8.va".

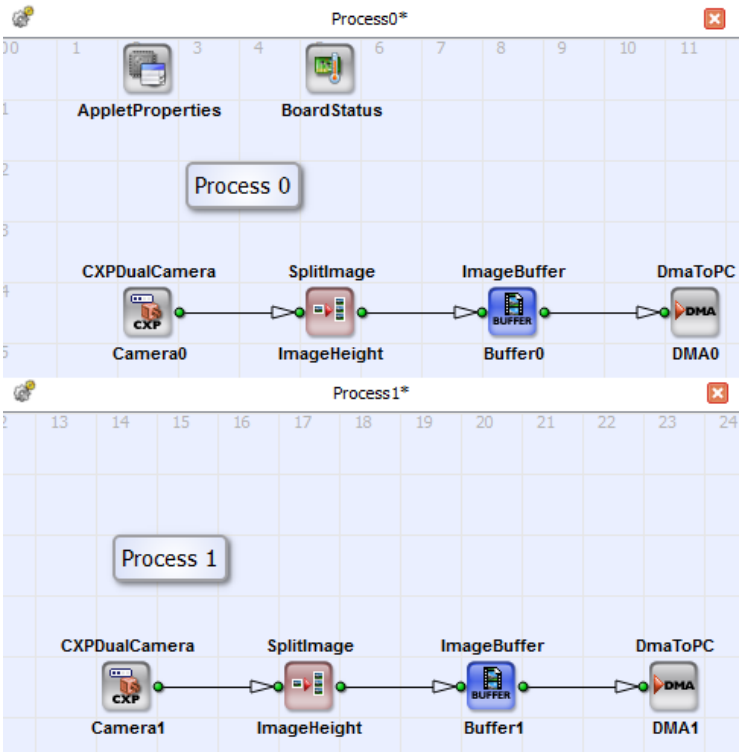



Figure 10.32. Basic Acquisition for RGB CoaxPress Line Scan Cameras in 6 Gbit/s Mode with Link Aggregation 2 on the ironman Frame Grabber

### 10.4.2.3. Basic Acquisition Examples for One Quad Line CoaXPress Line Scan Camera

"SingleCXP6x4LineGray8.va" (grayscale 8 bit) and "SingleCXP6x4LineRGB36.va" (RGB with 12 bits per color component) are example designs for one camera with 4 link aggregation. You can find the designs for the mE5VQ8-CXP6D and mE5-MA-VCX-QP platform under \examples\Acquisition \BasicAcquisition\mE5VQ8-CXP6D\Line and \examples\Acquisition\BasicAcquisition\mE5-MA-VCX-QP \Line. The designs for the two platforms differ but follow the same principle. In Fig. 10.33 you can see the basic design structure of the design "SingleCXP6x4LineGray8.va" for the mE5VQ8-CXP6D platform. As the memory bandwidth (128 bit RAM data width at 3.2 GB/s) for the 6 Gbit/s with link aggregation 4 is not sufficient with a single buffer, two buffers have to be used in parallel for the mE5VQ8-CXP6D platform. For the mE5-MA-VCX-QP platform only one DRAM element is necessary to achieve the required bandwidth of 6Gbit/s per line. Here up to 512 bit RAM data width at a RAM bandwidth of 12.8 GB/s are possible. Please read for more information on the devices resources Appendix A, '*Device Resources*'. In Fig. 10.34 you can see the basic design structure of the example "SingleCXP6x4LineGray8.va" on the mE5-MA-VCX-QP platform.

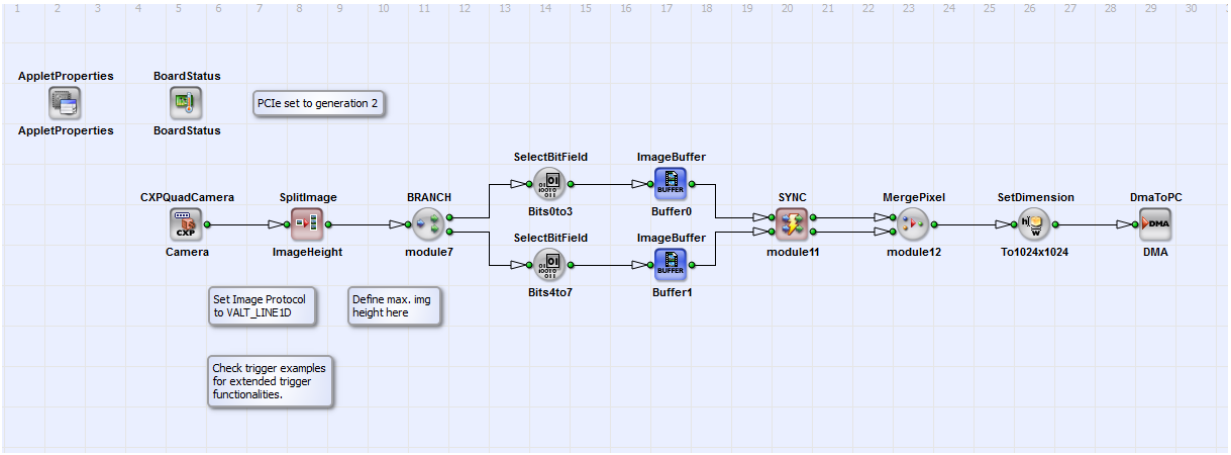



Figure 10.33. Basic Acquisition for Grayscale CoaxPress Line Scan Cameras in 6 Gbit/s Mode with Link Aggregation 4 on the ironman Frame Grabber

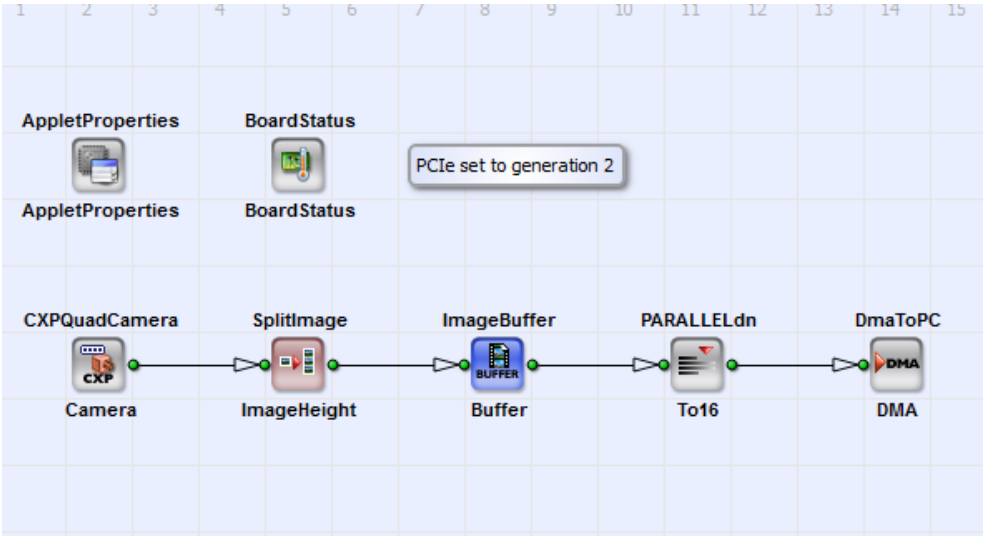



Figure 10.34. Basic Acquisition for Grayscale CoaxPress Line Scan Cameras in 6 Gbit/s Mode with Link Aggregation 4 on the ironman Frame Grabber

# 10.5. Basic Acquisition Examples for Cameras for CoaXPress 12 imaFlex Frame Grabber

In the following you can find basic acquisition examples for for the CoaXPress12 frame grabber imaFlex. In Section 10.5.1, 'CoaXPress Area Scan Cameras' designs for area scan cameras and in Section 10.5.2, 'CoaXPress Line Scan Cameras' designs for line scan cameras are presented.

## 10.5.1. CoaXPress Area Scan Cameras

In this section you find example implementations for basic acquisition for the CoaXPress 12 frame grabber imaFlex. In contrast to the camera interfaces of the microEnbale 5 series, only one camera operator exists for different CXP link aggregations. Via parameter *ConnectionCount*, you can set the number of CXP connections of the corresponding camera. Possibil values are x1,x2 and x4. The parallelism at the output link of the camera interface corresponds to the number of chosen CXP connections. At the output link of the camera operator *CxpCamera* on the imaFlex platform only the color format *VAF_Gray* and the bit width *8bit* is possible. In the example designs shown here, you can implement image acquisition designs, which support multiple bit widths, color formats and image protocols of the corresponding camera. The example designs are located at `\examples\Acquisition\BasicAcquisition\iF-CXP12-Q`.

### 10.5.1.1. Basic Acquisition Example for One CoaXPress12 Quad Link Area Scan Camera

The basic acquisition design for one quad link aggregation camera for the imaFlex platform is `SingleCXP12x4AreaGray8.va`. The following figure shows the basic design structure of the example design.

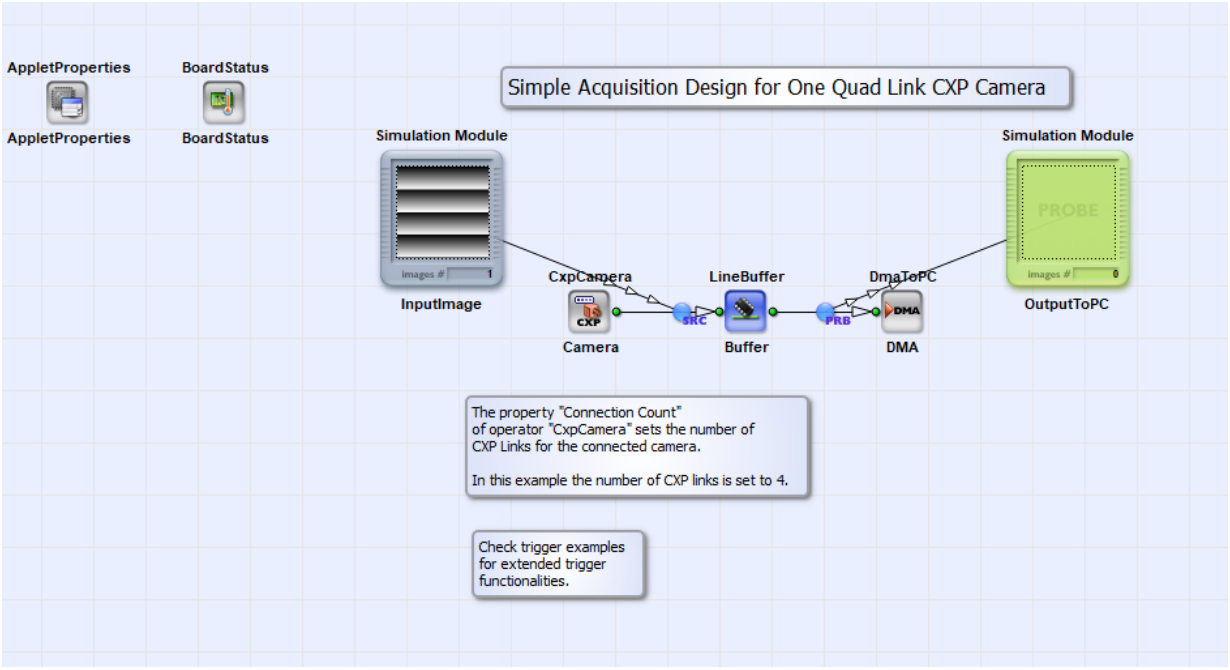



Figure 10.35. Basic Acquisition for One Grayscale CoaxPress Area Scan Camera with Link Aggregation 4 on the imaFlex Frame Grabber

### 10.5.1.2. Basic Acquisition Examples for Four CoaXPress-12 Single Link Area Scan Cameras

The example for four single link aggregation cameras is the design `QuadCXP12x1AreaGray8.va` (grayscale 8 bit). The shared memory concept is valid for the imaFlex platform. This information is highly relevant when you use multiple DRAM elements in one design. In the design and the *LineBuffer* documentation you find relevant information for the DRAM element *LineBuffer* on imaFlex in the context of maximum bandwidth performance: This operator performs an automatic aggregation of pixels for matching the memory data width as close as possible, in order to reach an optimal performance.

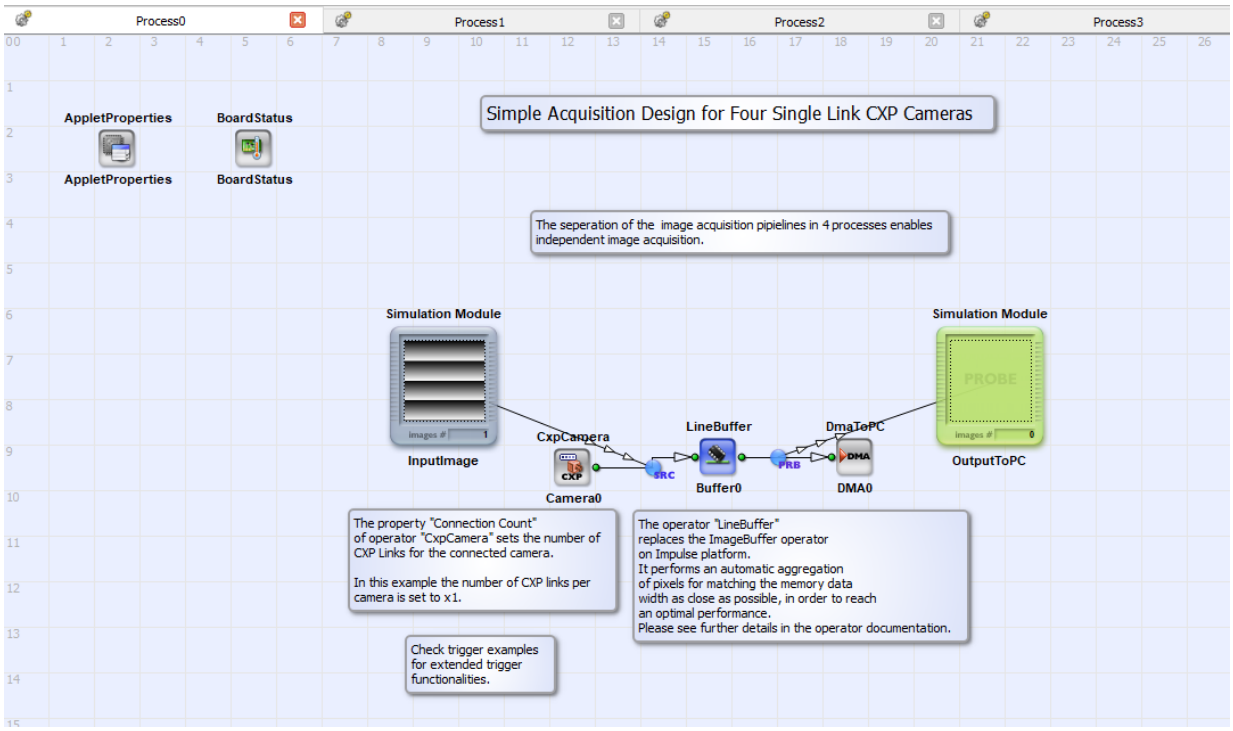



Figure 10.36. Basic Acquisition for Four CoaXPress12 Single Link Area Scan Cameras

### 10.5.1.3. Basic Acquisition Example for Multiple Bit Widths on imaFlex Platform

The example design `SingleCXP12x1Area_MultipleBitWidth.va` for the imaFlex platform demonstrates, how you can implement acquisition designs which support camera input bit widths of 8bit, 10bit, 12bit, 14bit and 16bit. The implementation differs from the microEnable 5 implementation, as on imaFlex platform the output link at the camera operator supports only 8-bit format. You find detailed comments and descriptions in the VisualApplets design. The following figure shows the basic acquisition design for the design `SingleCXP12x1Area_MultipleBitWidth.va`.

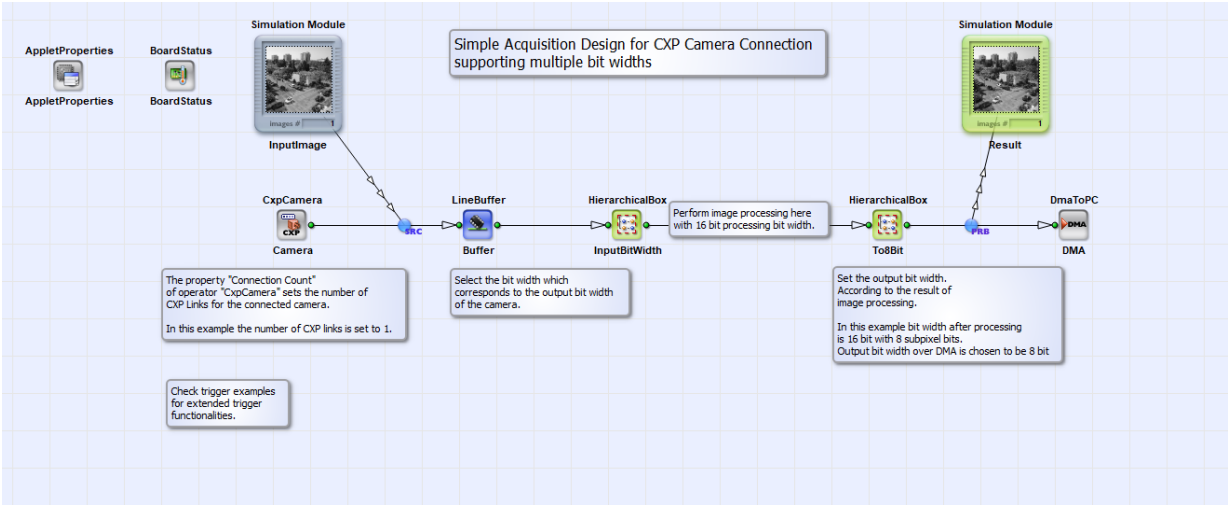



Figure 10.37. Basic Acquisition Example for Multiple Bit Widths on imaFlex Platform

## 10.5.1.4. Basic Acquisition Example for Color Format Support on imaFlex Platform

The example design `SingleCXP12x1Area_RGB.va` for the imaFlex platform demonstrates, how you can implement acquisition designs which support cameras with color format. The implementation differs from the microEnable 5 implementation, as on imaFlex platform the output link at the camera operator supports only grayscale color format. You find detailed comments and descriptions in the VisualApplets design. The following figure shows the basic acquisition design for the design `SingleCXP12x1Area_RGB.va`.

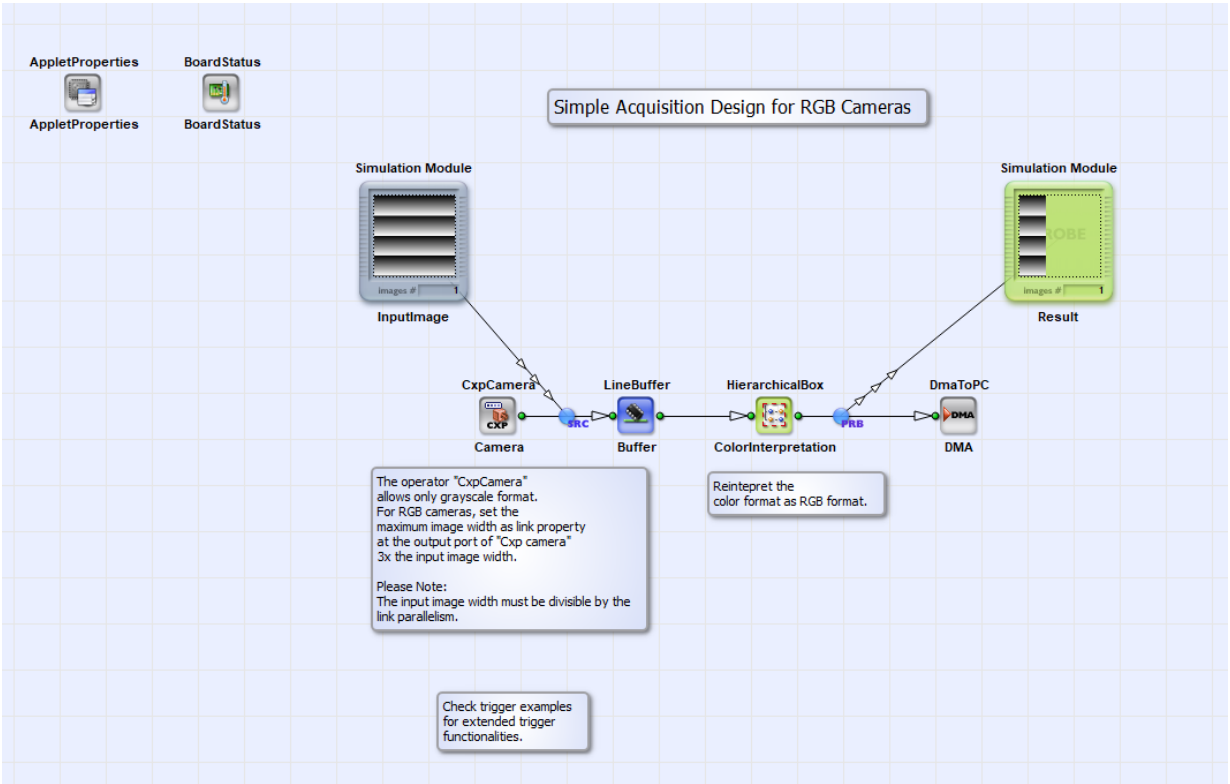



Figure 10.38. Basic Acquisition Example for Color Format Support on imaFlex Platform

## 10.5.2. CoaXPress Line Scan Cameras

The example presented in the following subsections is a basic acquisition design for grayscale line scan CoaXPress cameras. It is equivalent to the design in section Section 10.5.1, 'CoaXPress Area Scan Cameras'. The difference here is that the camera operator is set to VALT_LINE1D and add the operator *SplitImage* has been added.

### 10.5.2.1. Basic Acquisition Example for Single Line CoaXPress Line Scan Cameras

Find the basic acquisition designs `SingleCXP12x4LineGray8.va` (grayscale 8 bit) for one single link aggregation camera for the imaFlex platform at `\examples\Acquisition\BasicAcquisition\iF-CXP12-Q`. The following figure shows the basic design structure of the example design.

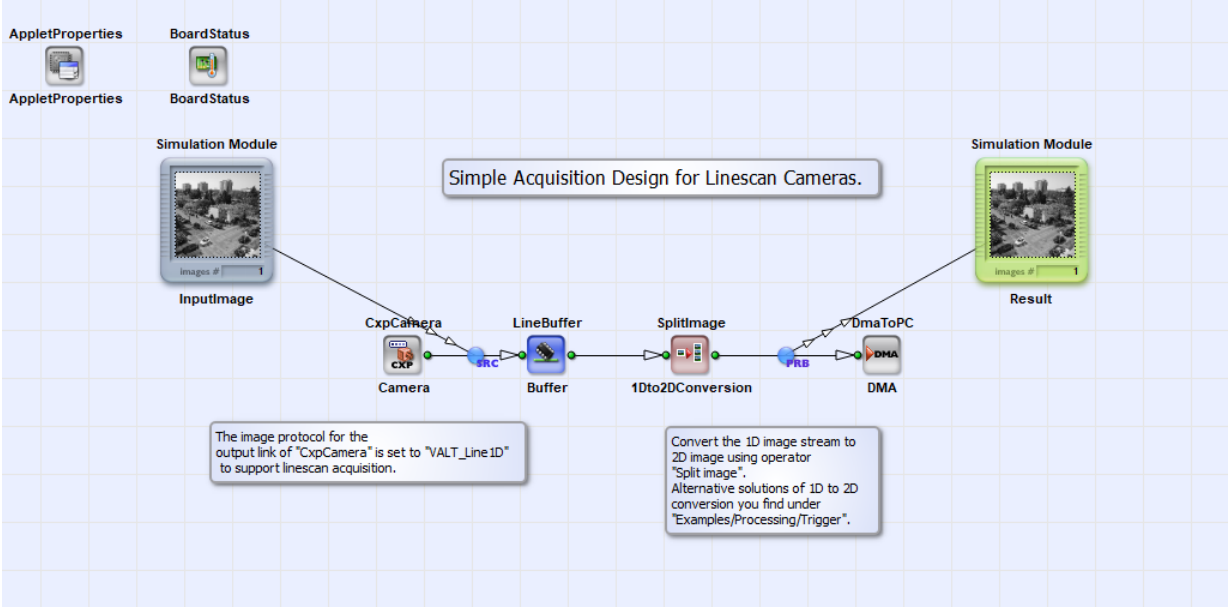



Figure 10.39. Basic Acquisition for Grayscale CoaxPress Line Scan Cameras on the imaFlex Frame Grabber

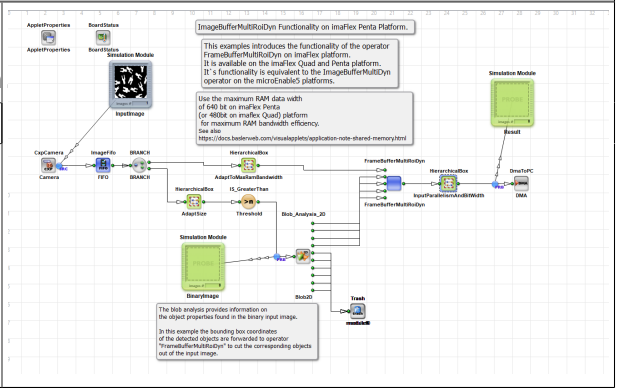
# 11. imaFlex CXP-12 Quad and Penta Implementation Examples

The imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms currently don't provide support for all DRAM-based memory operators, which are available on the microEnable 5 platforms. Operators which are not supported are: *ImageBufferMultiRoi* and *CoefficientBuffer*. In the VisualApplets example designs described in this chapter, you find user library elements and implementations with VisualApplets standard operators, which provide the same functionality as the missing memory operators. See Section 4.2.8, 'Delivered User Libraries' for instructions how to work with user library elements. In addition, the *FrameBufferMultiRoiDyn* operator is introduced, which substitutes the *ImageBufferMultiRoiDyn* operator on imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms.

The example "imaFlex_Jpeg_Color_iF_Advanced.va" demonstrates the usage of user library element *JPEG_Encoder_Color_iF* on imaFlex platform. Its functionality is equivalent to the functionality of the user lib elements of library JPEG_Color for microEnable 5 frame grabber series

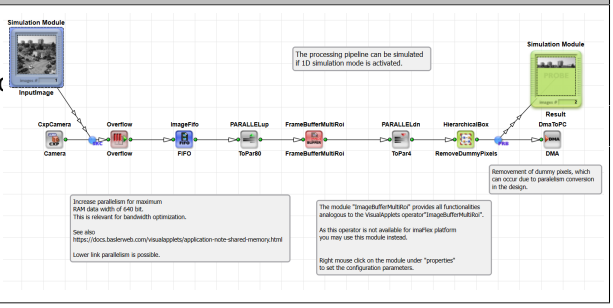
In Section 10.5, 'Basic Acquisition Examples for Cameras for CoaXPress 12 imaFlex Frame Grabber' basic acquisition examples are described, which show how to support multiple bit widths, image formats and protocols using the camera interface operator *CxpCamera* on imaFlex CXP-12 Quad platform. In section Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT', an example implementation is described, which shows how to perform 2D shading correction based on operator *RamLUT* as alternative to the operator *CoefficientBuffer*. The example Section 11.8, 'Example for the *DMAFromPC* Operator on the imaFlex CXP-12 Quad Platform' shows how images (in the example images containing shading correction coefficients) can be transferred from PC to the frame grabber using the *DMAFromPC* operator.

## 11.1. Functional Example for the *FrameBufferMultRoiDyn* Operator on the imaFlex CXP-12 Penta Platform

| **Brief Description** | |
|---|---|
| **Files:** `examples \imaFlex_ImplementationExamples \imaFlexPenta_CXP12_FrameBufferMultiRoiDyn` |  |
| **Default Platform:** imaFlex CXP-12 Penta | |
| **Short Description**<br><br>Demonstration of the functionality of the *FrameBufferMultiRoiDyn* operator. | |

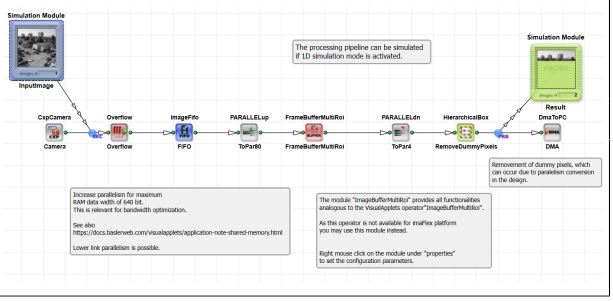
This example demonstrates how the *FrameBufferMultiRoiDyn* operator substitutes the functionality of the *ImageBufferMultiRoiDyn* operator on imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms.

## 11.2. Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Penta Platform

| Brief Description | |
|---|---|
| **Files:** examples \imaFlex_ImplementationExamples \imaFlexPenta_CXP12_FrameBufferMultiRoi.va |  |
| **Default Platform:** imaFlex CXP-12 Penta | |
| **Short Description**<br><br>Demonstration of the functionality of the *FrameBufferMultiRoi* user library element. | |

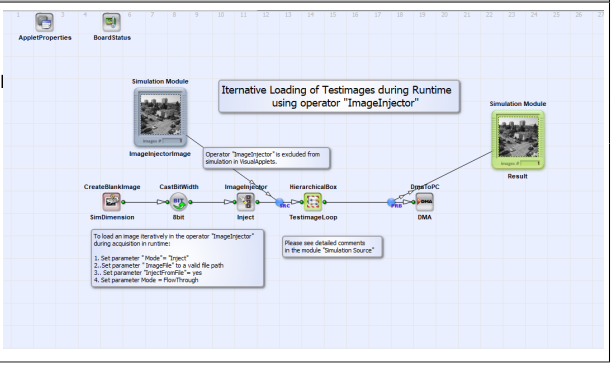
This example demonstrates how the *FrameBufferMultiRoi* user library element substitutes the functionality of the *ImageBufferMultiRoi* operator, which is not supported on the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms. See Section 4.2.8, 'Delivered User Libraries' for instructions how to work with user library elements. The functionality is equivalent to the functionality of the *ImageBufferMultiRoi* user library element, but uses less FPFA resources.

## 11.3. Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Quad Platform

| Brief Description | |
|---|---|
| **Files:** examples \imaFlex_ImplementationExamples \imaFlexQuad_CXP12_FrameBufferMultiRoi.vad |  |
| **Default Platform:** imaFlex CXP-12 Quad | |
| **Short Description**<br><br>Demonstration of the functionality of the *FrameBufferMultiRoi* user library element. | |

This example demonstrates how the *FrameBufferMultiRoi* user library element substitutes the functionality of the *ImageBufferMultiRoi* operator, which is not supported on the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms. See Section 4.2.8, 'Delivered User Libraries' for instructions how to work with user library elements. The functionality is equivalent to the functionality of the *ImageBufferMultiRoi* user library element, but uses less FPFA resources.

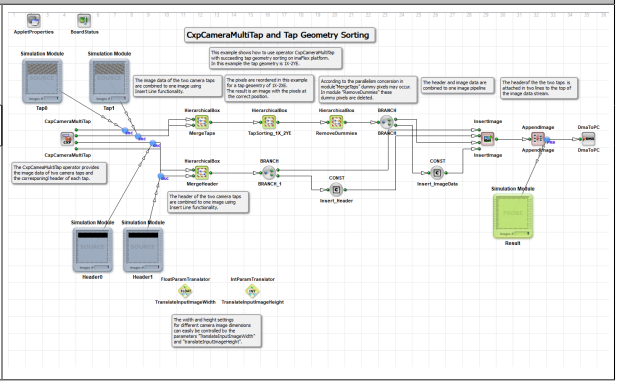
## 11.4. Functional Example for Loading Test Images Using *ImageInjector*

| Brief Description | |
|---|---|
| **Files:** examples \imaFlex_ImplementationExamples \imaFlex_CXP12_ImageInjector_SimulationSou |  |
| **Default Platform:** iF-CXP12-Q | |
| **Short Description**<br><br>Demonstration of loading test images during runtime using operator *ImageInjector* . | |

This example demonstrates how the test images can be loaded during runtime using the operator *ImageInjector*. The example implementation can substitute the functionality of the operator

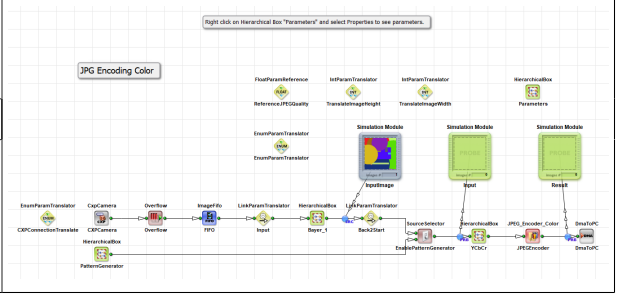
*CoefficientBuffer* as test image source on the imaFlex CXP-12 Quad platform. The operator *CoefficientBuffer* is not supported on the imaFlex CXP-12 Quad platform.

## 11.5. Functional Example for Multi Tap Camera Interface with Tap Geometry Sorting

| **Brief Description** | |
|---|---|
| **Files:** examples \imaFlex_ImplementationExamples \imaFlex_CXP12_CxpCameraMultiTap.vad<br>**Default Platform:** iF-CXP12-Q<br>**Short Description**<br>Demonstration of *CxpCameraMultiTap* succeeding with tap geometry sorting. |  |

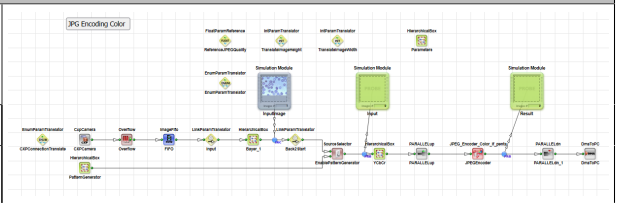
This example shows how to use operator *CxpCameraMultiTap* with succeeding tap geometry sorting on imaFlex CXP-12 Quad platform. In this example the tap geometry is 1X-2YE.

## 11.6. Functional Example for the *JPEG_Encoder_Color_iF* User Library Element on the imaFlex CXP-12 Quad Platform

| **Brief Description** | |
|---|---|
| **Files:** examples \imaFlex_ImplementationExamples \imaFlex_Jpeg_Color_iF_Advanced.vad<br>**Default Platform:** iF-CXP12-Q<br>**Short Description**<br>Demonstration of the functionality of the user library element *JPEG_Encoder_Color_iF*. |  |

This example demonstrates the usage of user library element *JPEG_Encoder_Color_iF* on the imaFlex CXP-12 Quad platform. Its functionality is equivalent to the functionality of the user lib elements of library JPEG_Color for microEnable 5 frame grabber series. See Section 4.2.8, 'Delivered User Libraries' for instructions how to work with user library elements.

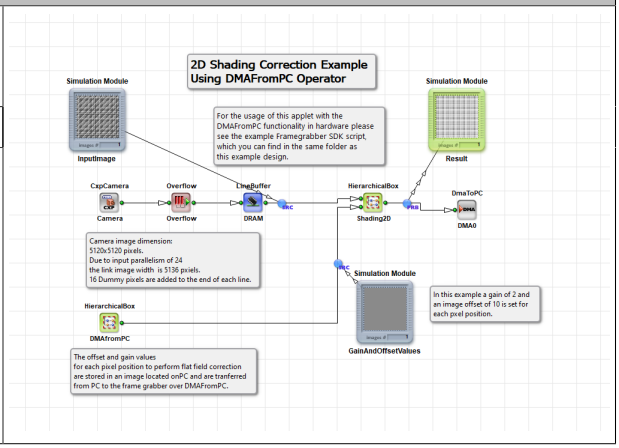
## 11.7. Functional Example for the *JPEG_Encoder_Color_iF_Penta* User Library Element on the imaFlex CXP-12 Penta Platform

| **Brief Description** | |
|---|---|
| **Files:** examples \imaFlex_ImplementationExamples \penta_Jpeg_Color_iF_Advanced.vad<br>**Default Platform:** iF-CXP12-Penta<br>**Short Description** |  |

| Brief Description | |
|---|---|
| Demonstration of the functionality of the user library element *JPEG_Encoder_Color_iF_Penta*. | |

This example demonstrates the usage of user library element *JPEG_Encoder_Color_iF_Penta* on the imaFlex CXP-12 Penta platform. Its functionality is equivalent to the functionality of the user lib elements of library JPEG_Color for microEnable 5 frame grabber series. See Section 4.2.8, 'Delivered User Libraries' for instructions how to work with user library elements.

## 11.8. Example for the *DMAFromPC* Operator on the imaFlex CXP-12 Quad Platform

| Brief Description | |
|---|---|
| **Files:** `examples \imaFlex_ImplementationExamples\DMAFromPC \DMAFromPC.vad` |  |
| **Default Platform:** imaFlex CXP-12 Penta | |
| **Short Description**<br><br>Demonstration of the functionality of the *DMAFromPC* operator. | |

This example demonstrates how you can use the *DMAFromPC* operator on an imaFlex CXP-12 Quad platform to transfer images from the PC to the frame grabber using the example of shading correction. See the Framegrabber SDK example script, which you can find in the same directory as the *.va file, to see how the *DMAFromPC* functionality can be used during runtime.

# 12. Processing Examples

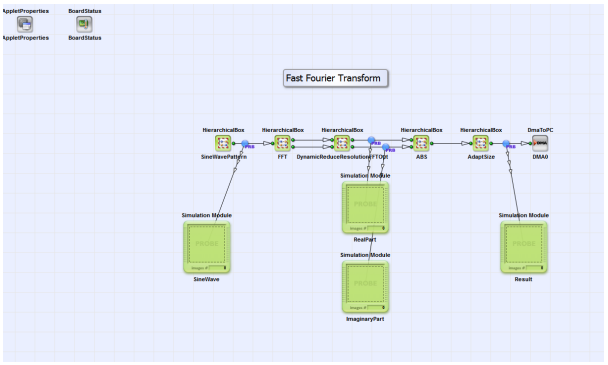
VisualApplets includes numerous examples which will show the usage of operators and will provide you with implementation ideas. All examples can be found in the VisualApplets installation directory in sub-directory examples i.e. "**%VASINSTALLDIR%/examples**". Testimages for simulation can be found in "**%VASINSTALLDIR%/testimages**"

The following sections show an overview on all examples. A brief introduction is presented as well as the folder and filename. All examples use a default hardware platform. Nearly most of them can easily be converted to other hardware platforms. See Section 4.5, 'Target Hardware Porting' for more information on how to switch the hardware platform.

## 12.1. Advanced

This section contains advanced examples on laser pointer detection and laser triangulation. Also examples for the usage of the operators **ImageBuffer_JPEG_Gray**, **JPEG_Encoder_Gray** and the fast fourier transformation operator **FFT** are provided. Additional examples show how lossless image compression can be performed based on run length encoding methods.

### 12.1.1. Fast Fourier transform

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Advanced \FastFourierTransform\mE5-MA-VCL\fft.va<br><br>**File:** \examples\Processing\Advanced \FastFourierTransform\mE5-VD8-CL\fft.va<br><br>**File:** \examples\Processing\Advanced \FastFourierTransform\iF-CXP12-Q\fft.va |  |
| **Default Platform:** mE5-MA-VCL or mE5-VD8-CL | |
| **Short Description**<br><br>Shows the usage of operator *FFT*. The applet generates a sine pattern and performs the FFT. The results show the frequency part. Use the VisualApplets simulation. | |

The provided example will demonstrate the use of the FFT. In the example, a simple sine wave with period 256 and magnitude ± 16384 is generated.

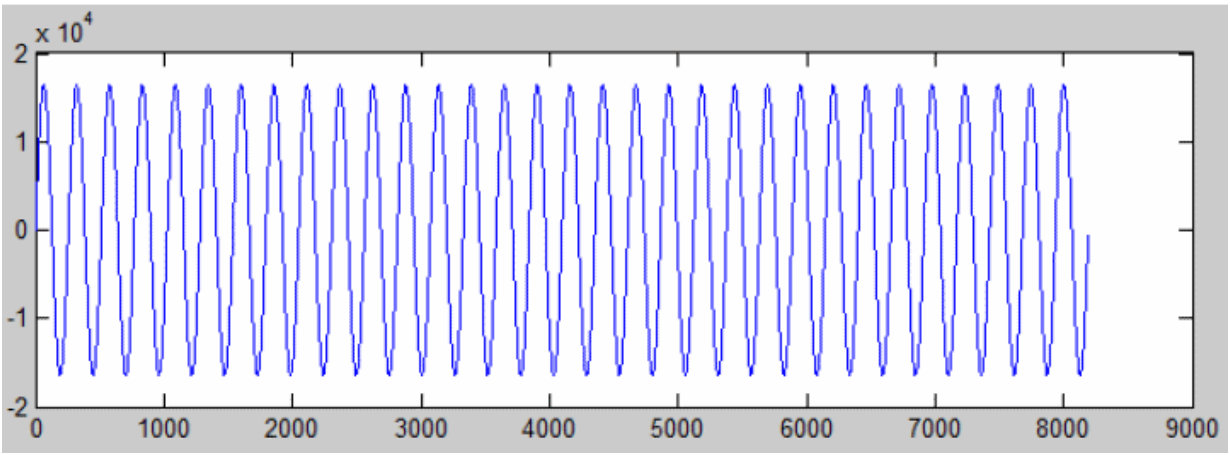



Figure 12.1. Sinewave

The resulting real and imaginary part are squared and added. So the output is the squared absolute value.

The result shows the expected peak at x = 32 i.e. a period of 8192 / 32 = 256. At position 32 the real part value is -43 and imaginary part is -67108738. When we compare the values with a Matlab FFT calculation we see real = 8,7e-10 and imag = 6710900. So we can see that the result is very good. There exists a small inaccuracy in the calculation only.

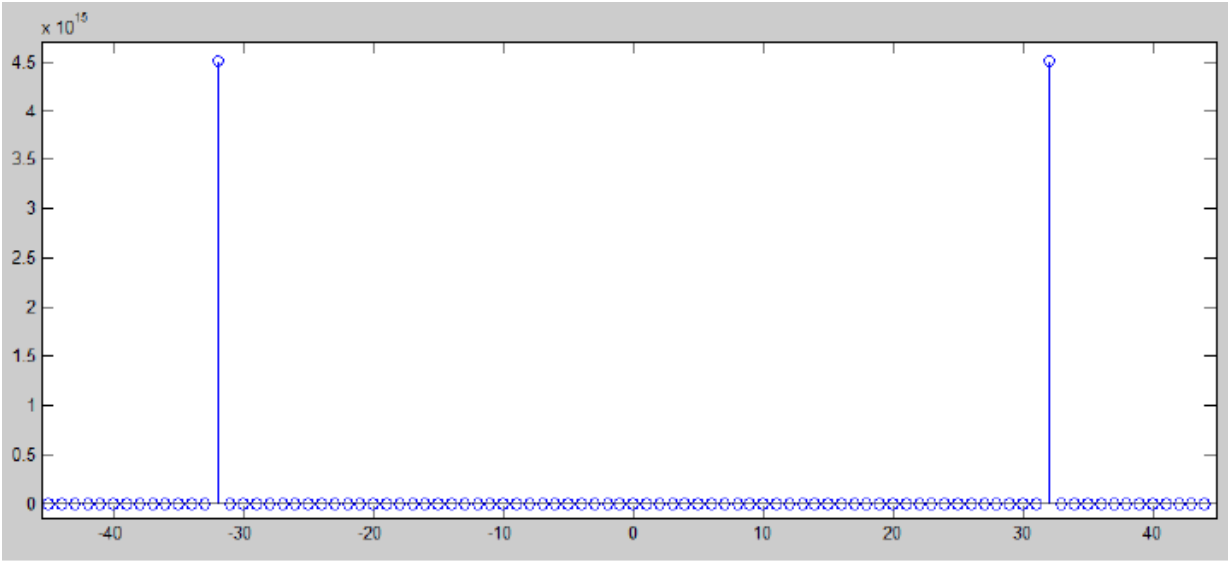



Figure 12.2. FFT Result

The implementation is very simple. A sine wave input. H-Box DynamicReduceResolutionFFTOut is used to shift results and reduce their resolution. This is not required for the given sine wave.

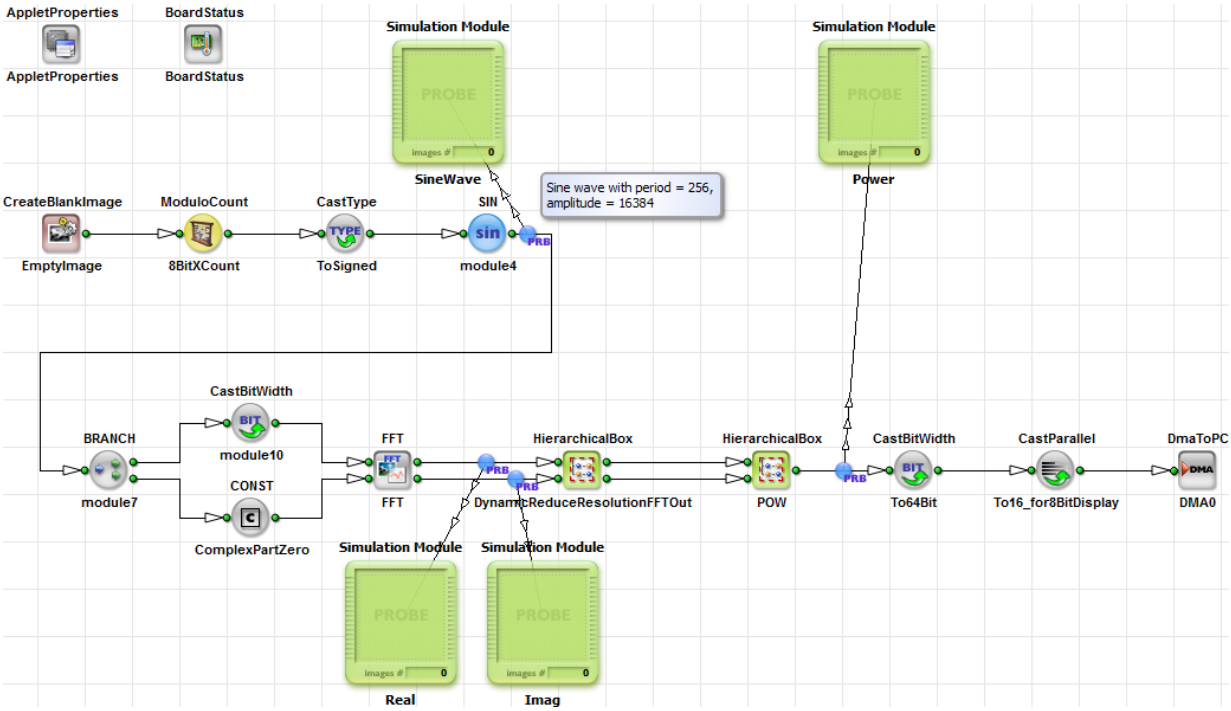
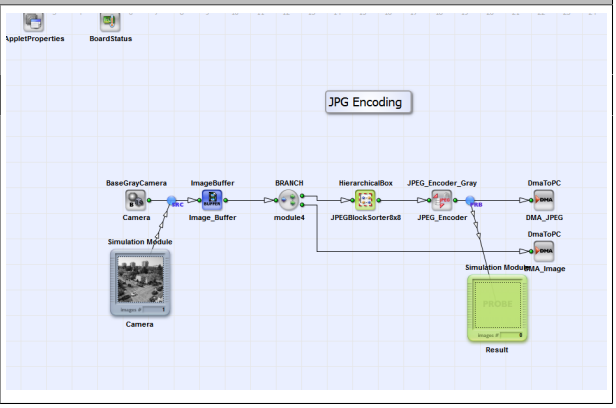


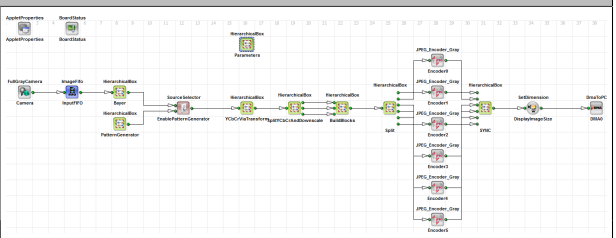

Figure 12.3. FFT Result

## 12.1.2. JPEG Encoder Gray

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Advanced\JPEG \JPEG_Gray\JPEG_Gray.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A simple example which shows the usage of the JPEG operators. | |



## 12.1.3. Using more than one JPEG encoder to enhance the bandwidth of JPEG compression.

| Brief Description | |
|---|---|
| **Files:**<br>\examples\Processing\Advanced\JPEG \JPEG_multi\JPEG_Color.va<br>\examples\Processing\Advanced\JPEG \JPEG_multi\JPEG_Gray.va<br>\examples\Processing\Advanced\JPEG \JPEG_multi\Software |  |
| **Default Platform:**<br>mE5-MA-VCL<br>mE5-MA-ACX-QP | |
| **Short Description**<br><br>Using more than one encoder to enhance the bandwidth for a JPEG compression. | |

This example is designed to enhance the bandwidth for a JPEG compression. To allow this several encoders are used in parallel. There exist two examples for the marathon mE5 VCL plattform. The JPEG_Gray example uses 4 encoders to encode a full configuration grayscale image. The JPEG_Color example uses 6 encoders to encode an image from a bayer camera with a subsampling in horizontal and vertical direction for the Chroma components.

### 12.1.3.1. JPEG_Gray - VisualApplets Design

This Design uses four JPEG_Encoder_Gray operators to to encode an image with a bandwidth of maximum 1200MP/s (this would be suffizient for a CameraLink Camera in full configuration).
A JPEG Stream bases on a runlength encoding. Each Stream starts with a DC part followed by several AC components. So in order to split an image to several encoders the encoding needs to be restarted in intervals. This can be done in the JPEG format using Restart markers. The VisualApplets design therefor splits the Image into Blocks of 8 Lines (1 Line of MCUs). At the end of each MCU Line a restart Marker is inserted.

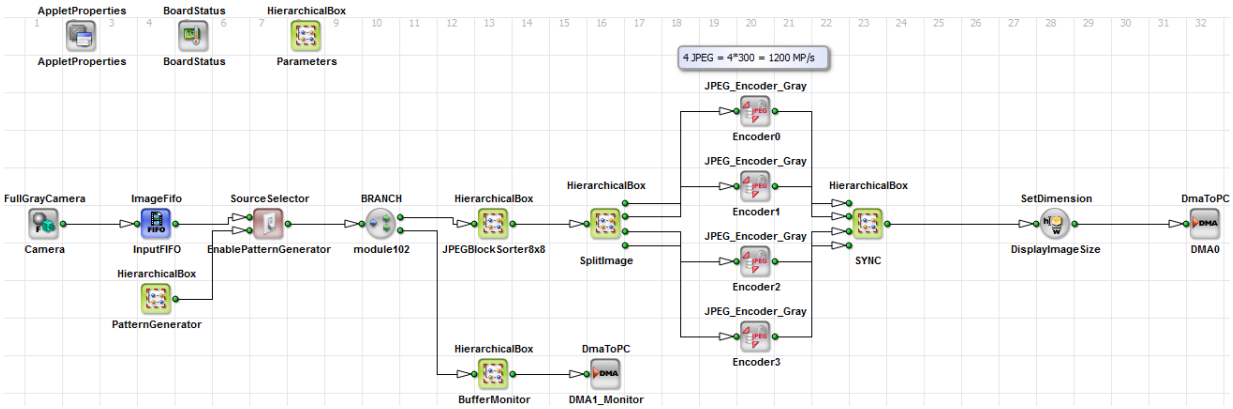



Figure 12.4. Basic design structure

In Fig. 12.4 you can see the general applet. For the compression there are three interesting hierarchical boxes (**JPEGBlockSorter8x8**, **SplitImage** and **SYNC**). In **JPEGBlockSorter8x8**(Fig. 12.5) the data of 8 lines is rearanged to blocks of 8x8 pixel. These blocks are send consecutively.
To keep the image handy the dimensions are kept the same.
**SplitImage**(Fig. 12.7) saves the image to the DRAM and rearanges the lines in a way, that the 4 consecutive lines addresses 4 independend MCU-block-lines. In this way for streams of MCU-block-lines can be split and send to 4 independend encoders.
**SYNC** collects the encoded streams, replaces the information Tag at the end by a restart marker and appends all lines to an image.

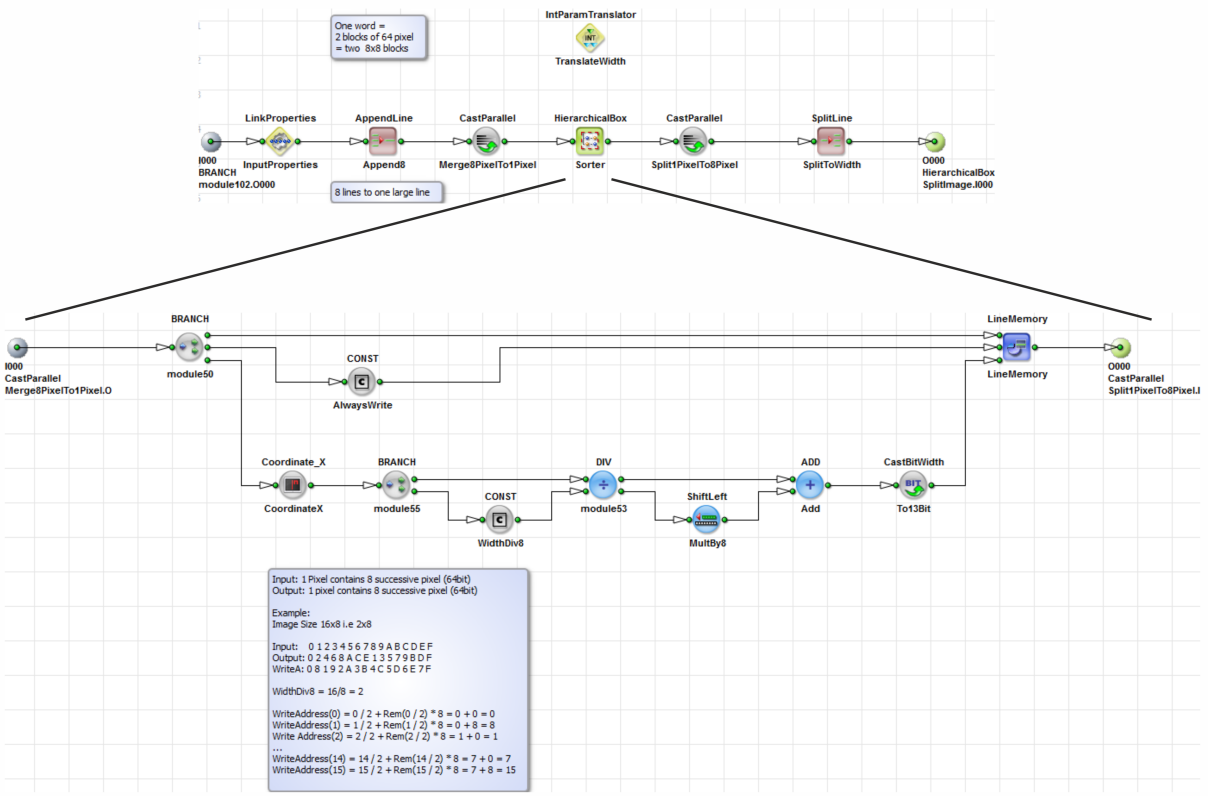



Figure 12.5. Content of box **JPEGBlockSorter8x8**

First 8 pixel are appended to one "block-pixel". In the Sort Box these "block-pixel" are arranged in a way that the block pixel of each line follow after another as shown in Fig. 12.6

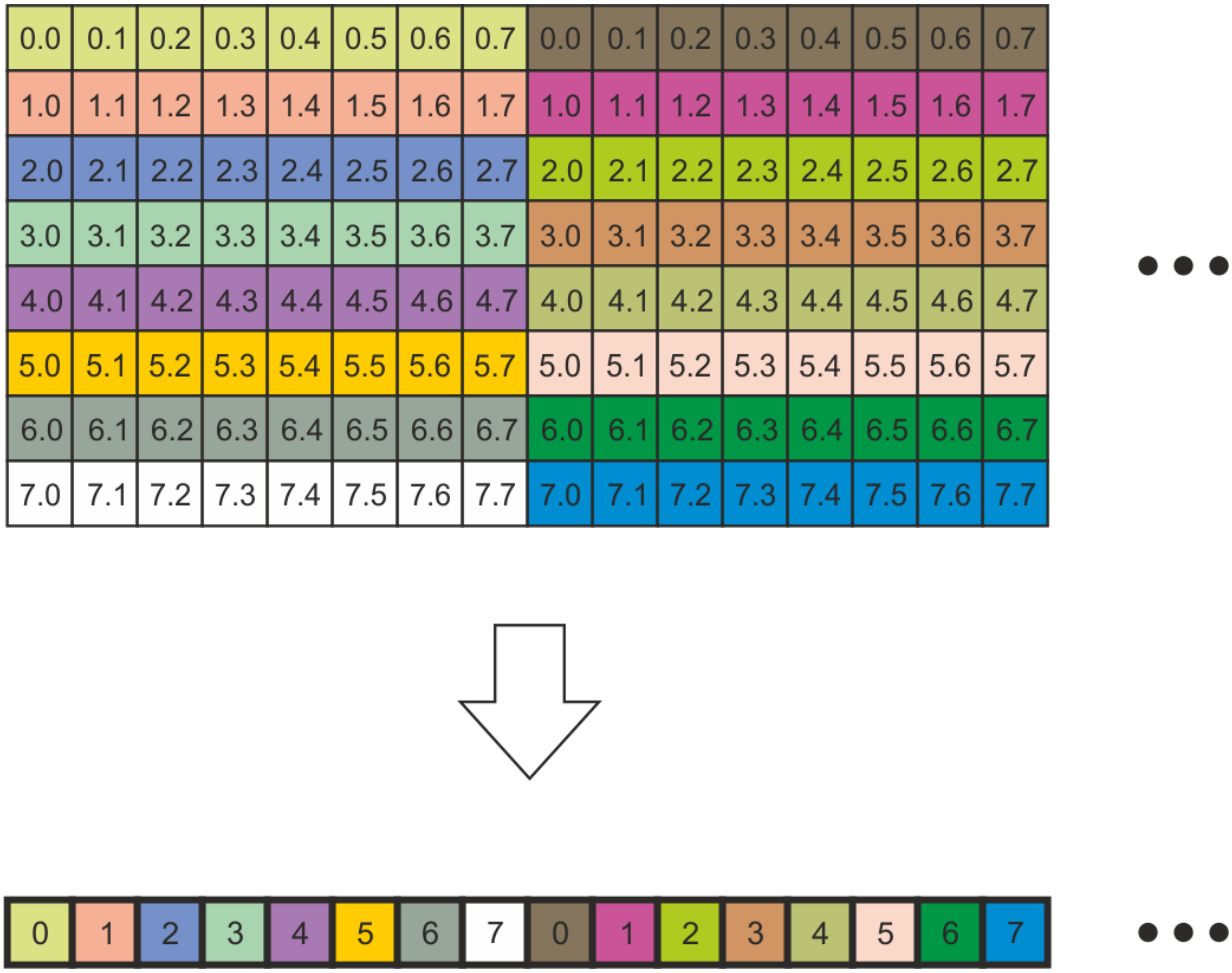



Figure 12.6. Rearangement of Pixel in JPEGBlockSorter8x8

The **SplitImage**(Fig. 12.7) box rearanges the lines from the **JPEGBlockSorter8x8**(Fig. 12.5) into a schema, where always 4 blocks are separatly transmitted. Each Block consists of 8 Lines. This procedure is repeated till all lines are transmitted.
In the **Split** box the image is split into 4 Images. This is done by removing all lines from each link that doesn't belong to the image. These sub images are split to images of the height 8, so that the applet can insert restart intervalls at the end of 8 lines.

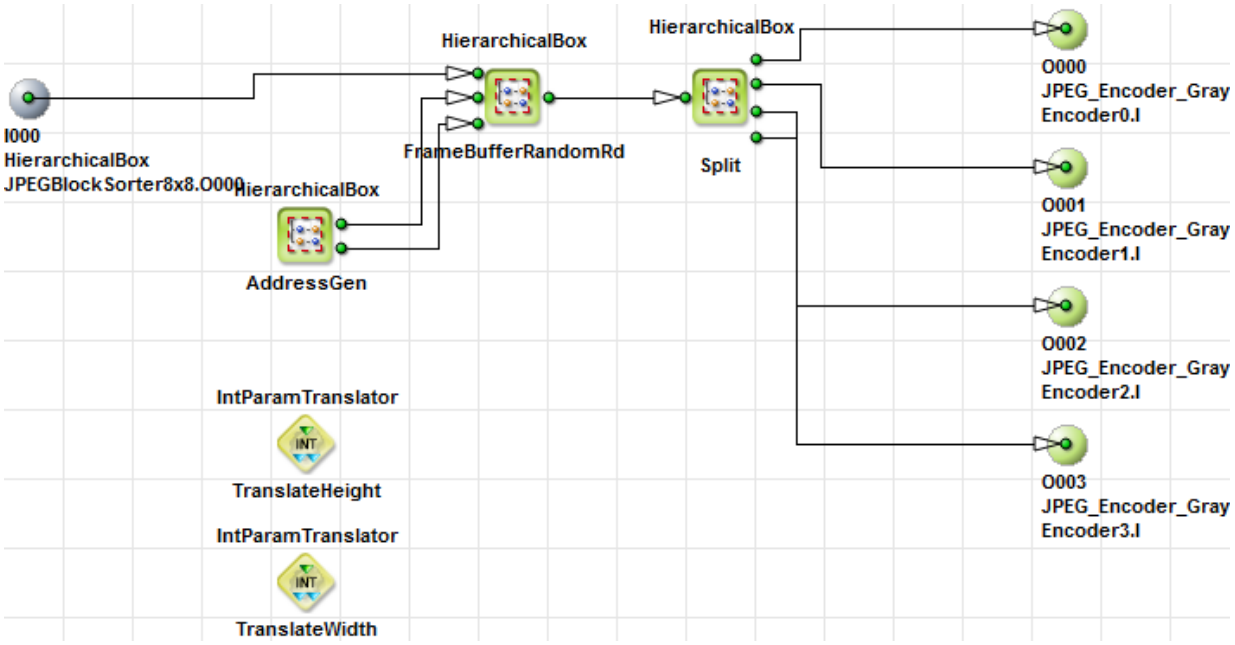



Figure 12.7. Content of **SplitImage** box

The **SYNC** box combines the streams of each JPEG_Converter into one image. This is done in three steps. First the final bytes (0xFFD9 - End of Image Marker + Informations from the operator) are marked for removal. This is done in the Box **RemoveFillByte**(Fig. 12.8). Second restart markers are added in between the blocks. Restart Markers in a JPEG stream are in a structure: "0xFFDx", where x is a value from 0 to 7 counting round and robbin. This is done in the Box **RestartMarker**(Fig. 12.5). Third unused Bytes are removed.

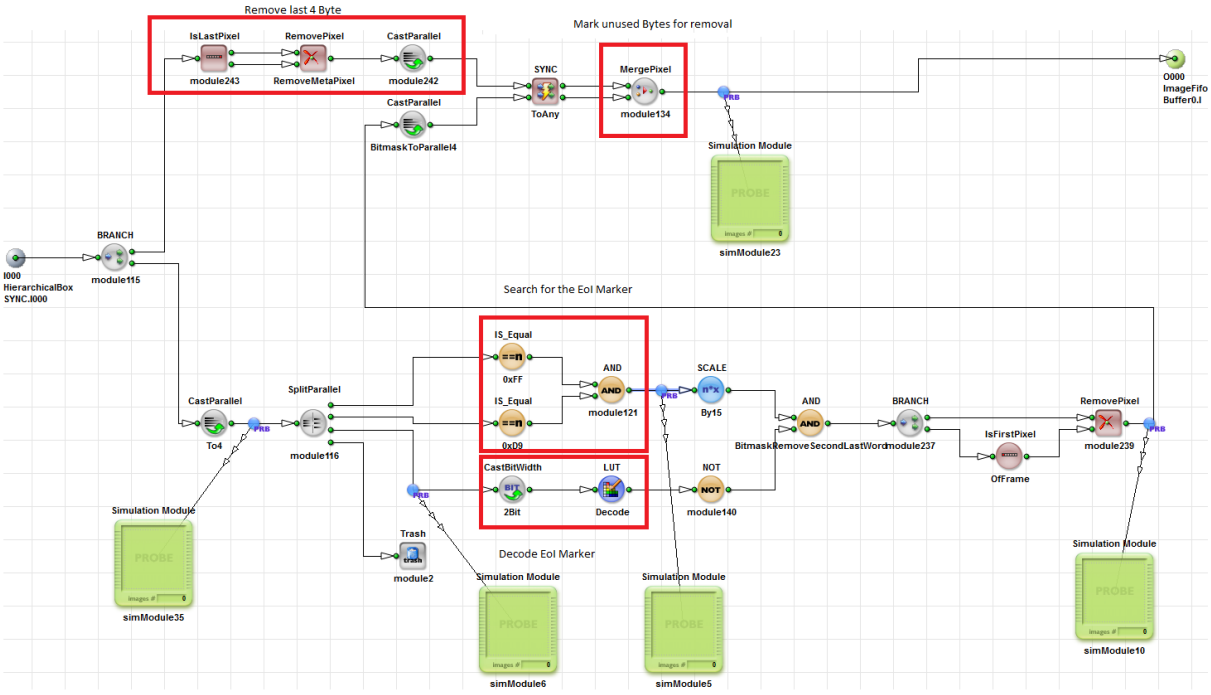



Figure 12.8. Content of the **RemoveFillByte** box

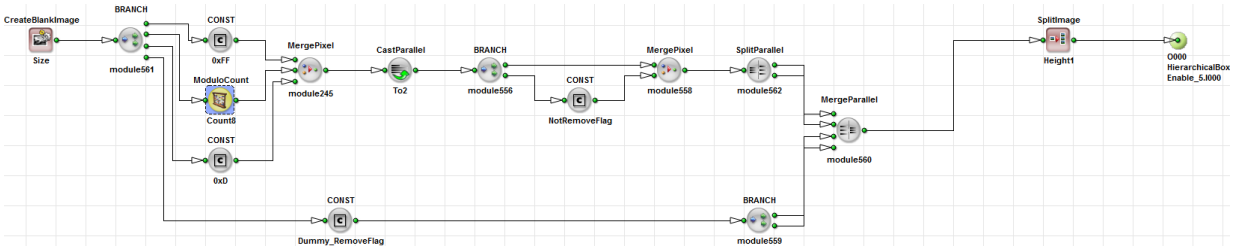



Figure 12.9. Content of the **RestartMarker** box

For the imagedata you get out of this applet you need to add Restart information to the header. This is done by adding the DRI Marker (0xFF DD) to the header.

## 12.1.3.2. JPEG_Gray - VisualApplets Design

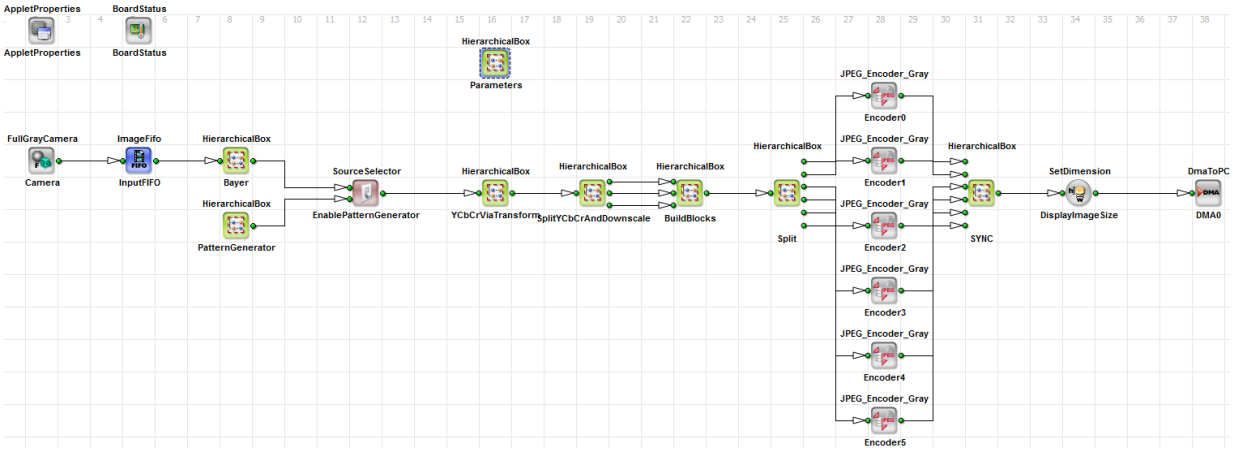
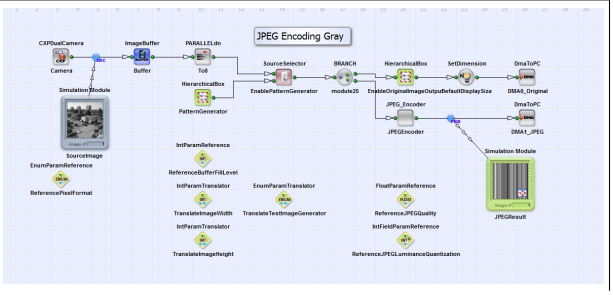



Figure 12.10. Basic design structure

This Example is basically the same as the Gray example. The differences are:
1. RGB data from a Bayer conversion is converted to the YCrCb colorspace.
2. Cr and Cb components are subsampled.
3. The separated images are buffered in a separate buffer. In order to sort the Lines to 6 converters without inserting dummy lines an independend sorting buffer is needed.
4. Sync needs to be split into 3 different images of different height.

In order to use restart interfalls in a subsampled image the sampling factor needs to be adjusted to the minimum number of MCUs in a channel. We use a 4:2:0 Subsampling. This means we have only half the line width in the Cr and Cb image. That's why lines need to be split in half in the Y image as well in order to get all restart markers in the same place.

## 12.1.4. JPEG Compression Using Operator JPEG_Encoder

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Advanced\JPEG\mE5-MA-VCL\JPEG_DualBaseAreaGray.va<br>\examples\Processing\Advanced\JPEG\mE5-MA-VCL\JPEG_SingleFullAreaGray.va<br>\examples\Processing\Advanced\JPEG\mE5-MA-VCX-QP\JPEG_SingleCXP6x4AreaGray.va<br>\examples\Processing\Advanced\JPEG\mE5-MA-VCX-QP\JPEG_DualCXP6x2AreaGray.va |  |

| Brief Description | |
|---|---|
| \examples\Processing\Advanced\JPEG\mE5-MA-VCX-QP\JPEG_QuadCXP6x1AreaGray.va | |
| **Default Platform:** mE5-MA-VCL mE5-MA-VCX-QP | |
| **Short Description**<br><br>Simple examples which show the usage of the operator **JPEG_Encoder**. | |

In these VisualApplets examples we show the usage of the operator **JPEG_Encoder** for the compression of grayscale 8 bit images for mE5-MA-VCL and mE5-MA-VCX-QP platform for various link configurations and input bandwidths. For news on the operator **JPEG_Encoder**, the example designs or if you have questions please have a look in our VisualApplets forum under https://forum.silicon.software/forum/index.php?thread/100-new-high-speed-jpeg-operator-and-examples-va-3-2-0/ .

### 12.1.4.1. Grayscale JPEG Encoding

In Fig. 12.11 you can see the top level design of a process of the VisualApplets examples for grayscale JPEG encoding with the modules **Implementation** and **Parameters**. In this top level design you find also information on the number of processes in the design and on the design clock frequency used in the example implementation.

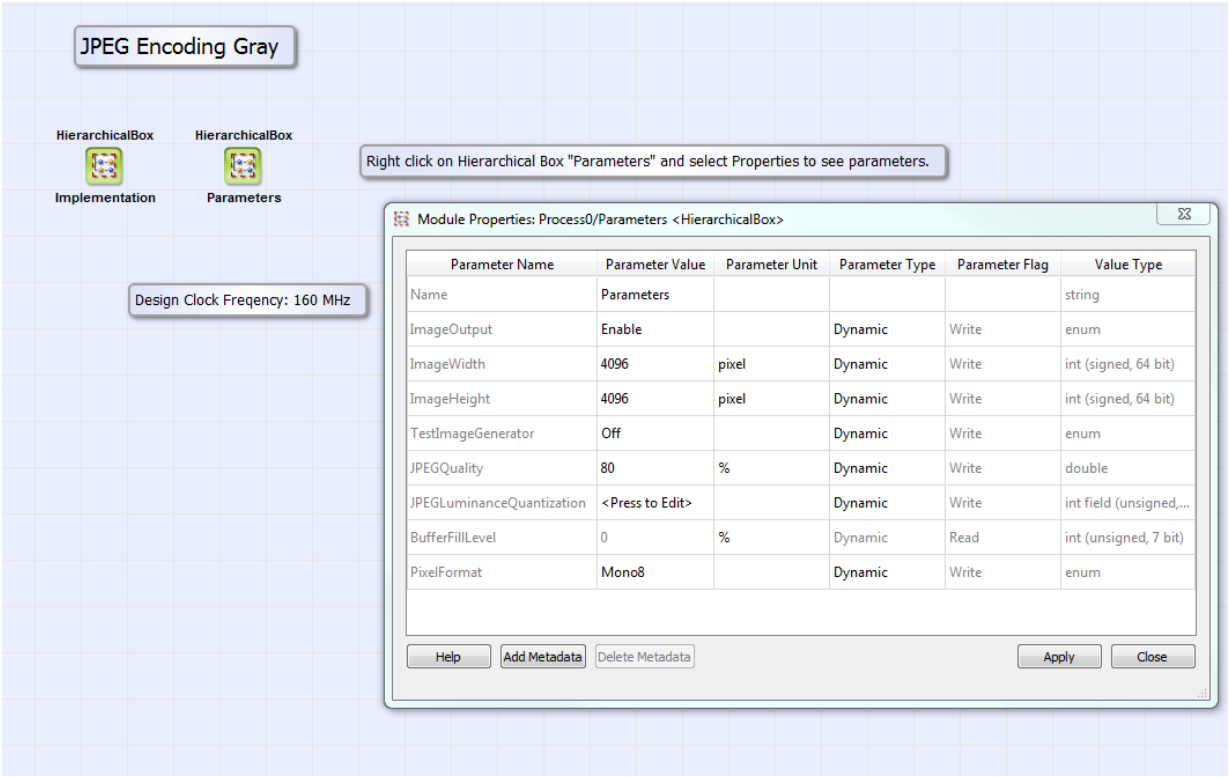



Figure 12.11. Top level design structure

With right-mouse-click on the box **Parameters** you can set the applet parameters like image dimension, JPEG quality or pixel format. You can set these parameters dynamically during execution of the applet in hardware. In the module **Implementation** the JPEG compression is implemented. Please see Fig. 12.12 for the basic implementation structure.

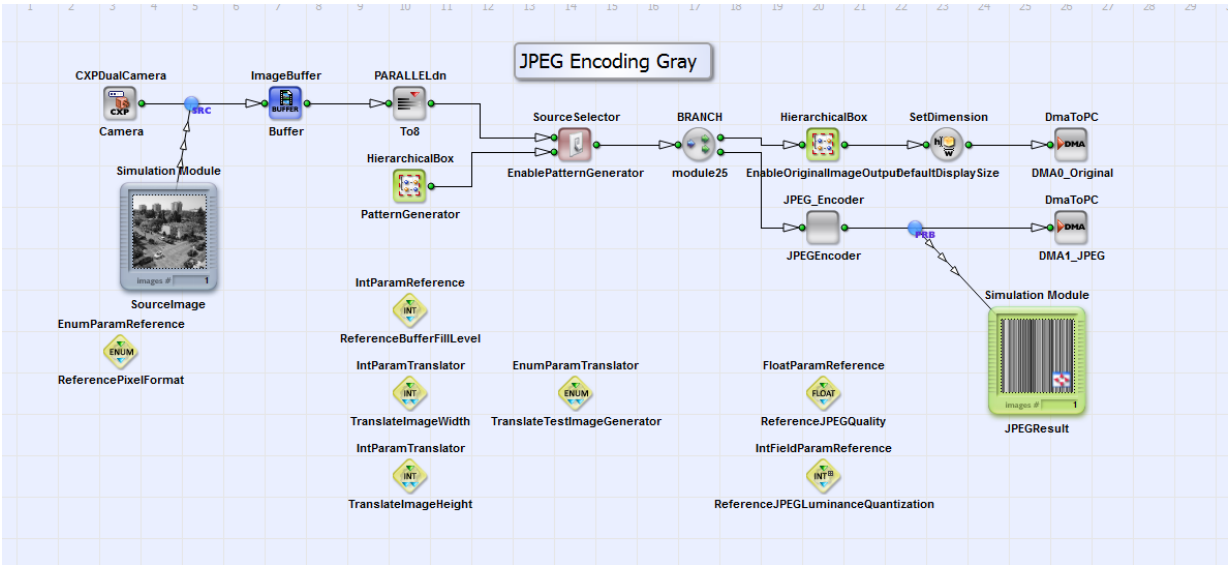



Figure 12.12. Basic implementation of grayscale JPEG compression using operator **JPEG_Encoder**

An grayscale 8 bit image with maximum image dimensions of 5120x5120 pixels is forwarded from camera interface directly to operator **JPEG_Encoder**. In comparison to previous JPEG operator **JPEG_Encoder_Gray** no external 8x8 pixels block sorting module is necessary. With right-mouse-click on the operator you can set parameters like image width and height or you can include or exclude a JPEG header. The quality of the JPEG compression can be set using two options: You can set the quality of the compression in percent using parameter **Quality** or you can configure the quantization table values individually in setting the values of the matrix parameter **LuminanceQuantization**. As alternative you can modify these parameters in the top level design with right-mouse-click on module **Parameters** using the corresponding translate and reference operators (see Fig. 12.12). The compressed JPEG stream is then transmitted via **DMA1_JPEG** to PC. Via **DMA0_Original** the user has the possibility to transfer the original input image to PC. This can be realized in setting the parameter **ImageOutput** in the Properties library of module **Parameters** in the top level design (see above) to "Enable". A disabling of the transfer of the original image via DMA minimizes the DMA payload. In the design the user has the possibility to transmit images generated by a pattern generator instead of images acquired by a camera. This can be done in choosing the input source of Source Selector operator **EnablePattern Generator** or in setting the corresponding parameter **TestImageGenerator** in the module **Parameters** in the top level design.

### 12.1.4.2. Design Versions

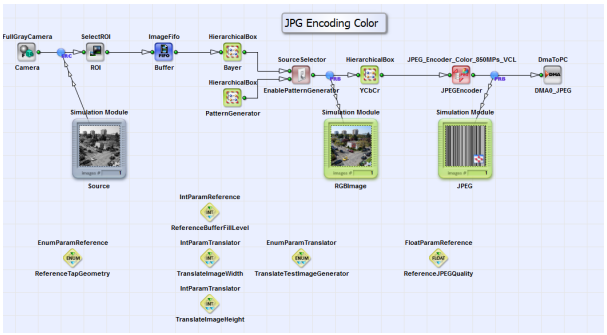
For JPEG grayscale encoding five example designs for microEnable 5 marathon platform for CameraLink and CoaxPress camera interfaces and for different link configurations and bandwidths are implemented. With the operator **JPEG_Encoder** the full data rate defined by the input link parameters can be processed. The bandwidth which can be processed in a design is defined by the by the design clock frequency and the link paraellism. Please see the following table for the available design versions:

| Example | Camera Interface | Number of Processes | Input Bandwidth per Camera Interface |
|---|---|---|---|
| JPEG_DualBaseAreaGray.va | Camera Link Base (mE5-MA-VCL) | 2 | Camera Link Base Speed 255 MP/s: Parallelism 4, Design Clock Frequency: 125 MHz |
| JPEG_SingleFullAreaGray.va | Camera Link Full (mE5-MA-VCL) | 1 | Camera Link Full Speed 850MP/s: Parallelism 8, Design Clock Frequency: 125 MHz |
| JPEG_SingleCXP6x4AreaGray.va | CoaxPress Quad Link (mE5-MA-VCX-QP) | 1 | 2500 MPixel/s: Parallelism 16, Design |

| Example | Camera Interface | Number of Processes | Input Bandwidth per Camera Interface |
|---|---|---|---|
| | | | Clock Frequency: 160 MHz |
| JPEG_DualCXP6x4AreaGray | CoaxPress Dual Link(mE5-MA-VCX-QP) | 2 | 1280 MPixel/s: Parallelism 8, Design Clock Frequency: 160 MHz |
| JPEG_QuadCXP6x1AreaGray | CoaxPress Single Link(mE5-MA-VCX-QP) | 4 | CoaxPress Single Link Speed 780 MPixel/s: Parallelism 8, Design Clock Frequency: 125 MHz |

Table 12.1. Design Versions for Grayscale JPEG Encoding

### 12.1.5. JPEG Color Compression Using User Library Elements

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Advanced\JPEG\mE5-MA-VCL\JPEG_DualBaseAreaBayer.va<br>\examples\Processing\Advanced\JPEG\mE5-MA-VCL\JPEG_SingleFullAreaBayer.va<br>\examples\Processing\Advanced\JPEG\mE5-MA-VCX-QP\JPEG_SingleCXP6x4AreaBayer.va<br>\examples\Processing\Advanced\JPEG\mE5-MA-VCX-QP\JPEG_DualCXP6x2AreaBayer.va<br>\examples\Processing\Advanced\JPEG\mE5-MA-VCX-QP\JPEG_TripleCXP6x1AreaBayer.va |  |
| **Default Platform:** mE5-MA-VCL<br>mE5-MA-VCX-QP | |
| **Short Description**<br><br>Simple examples which show the usage of the user library elements **JPEG_Encoder_Color_300MPs_VCL**, **JPEG_Encoder_Color_850MPs_VCL**, **JPEG_Encoder_Color_600MPs_VCX**, **JPEG_Encoder_Color_800MPs_VCX** and **JPEG_Encoder_Color_2500MPs_VCX**. | |

In these VisualApplets examples we show the usage of the user library elements **JPEG_Encoder_Color_300MPs_VCL**, **JPEG_Encoder_Color_850MPs_VCL** and **JPEG_Encoder_Color_2500MPs_VCX** for the compression of color images for mE5-MA-VCL and mE5-MA-VCX-QP platform for various link configurations and input bandwidths. For news on these user library elements, the corresponding design examples or if you have questions please have a look in our VisualApplets forum under https://forum.silicon.software/forum/index.php?thread/100-new-high-speed-jpeg-operator-and-examples-va-3-2-0/ .

#### 12.1.5.1. Color JPEG Encoding

In Fig. 12.13 you can see the top level design for each process of the VisualApplets examples for color JPEG encoding with the modules **Implementation** and **Parameters**. In this top level design you find also information on the number of processes in the design and on the design clock frequency used in the example implementation.

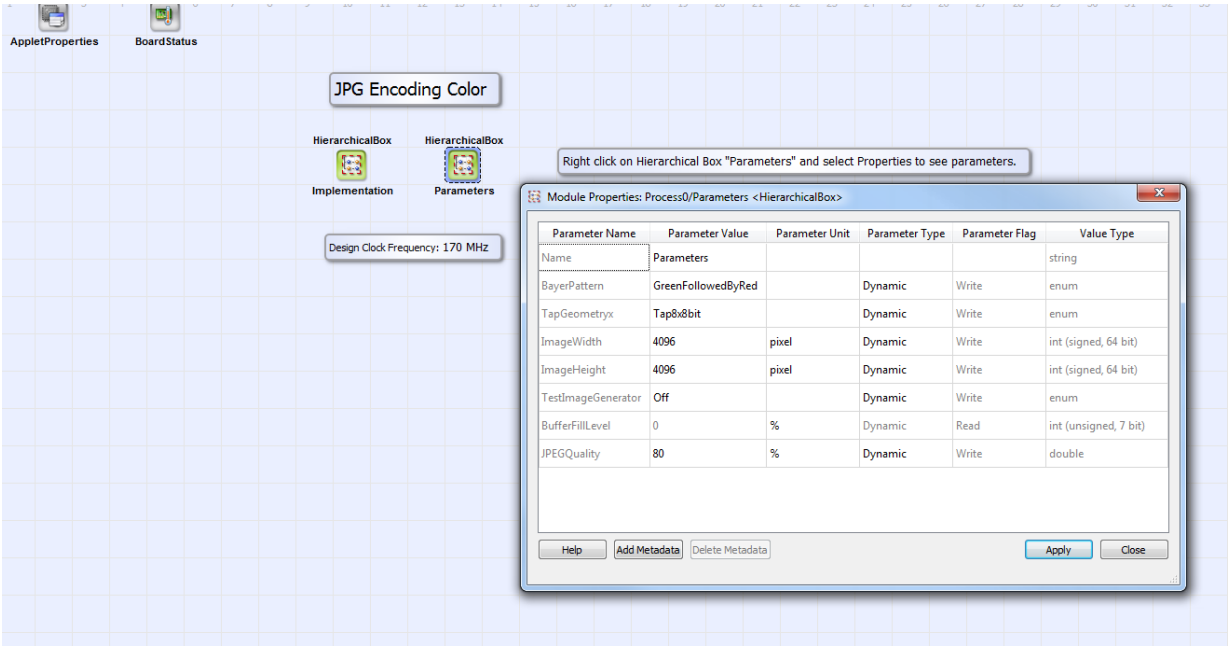



Figure 12.13. Top level design structure

With right-mouse-click on the box **Parameters** you can set the applets parameters like image dimension, JPEG quality, pixel format or Bayer Pattern. You can set these parameters dynamically during execution of the applet in hardware. In the module **Implementation** the JPEG compression is implemented. Please see Fig. 12.14 for the basic implementation structure.

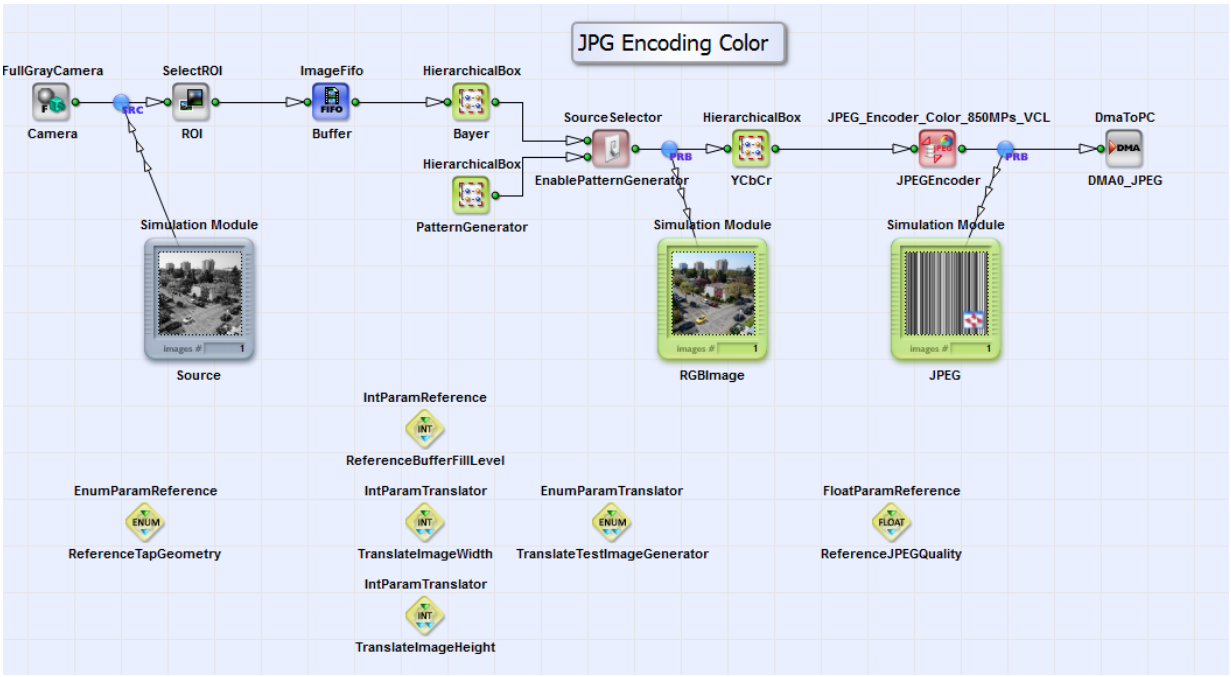



Figure 12.14. Basic implementation of color JPEG compression using user library elements

The input images acquired by a Bayer camera with maximum image dimensions of 5120x5120 pixels are forwarded to the module **Bayer**. In this module the color images are reconstructed using the DeBayering operator **Bayer3x3Linear**. Output of this module are RGB color images with 8 bit per color component. These images are converted from RGB to YCbCr color space using module **YCbCr** before they are forwarded to the user library elements for color JPEG compression. The color JPEG compression is performed using a chroma subsampling of 4:2:0. With right-mouse-click

on the user library element you can configure parameters like image width and height (maximum image dimensions: 5120x5120 pixels) or the quality of the JPEG compression in percent. As a more comfortable alternative you can modify these parameters in the top level design with right-mouse click on module **Parameters** (see above). The compressed JPEG Huffman stream with included header is then transmitted via **DMA0_JPEG** to PC. Each of the user elements for color JPEG compression is optimized for a specific platform and can process an individual maximum bandwidth. The name of the user library elements gives hint to these properties. In the following section you find a table of the user library elements used in the example designs for color JPEG compression. In the designs the user has furthermore the possibility to process images generated by a color pattern generator instead of images acquired by a camera. This can be realized in choosing the input source for Source Selector operator **EnablePatternGenerator** or in setting the corresponding parameter **TestImageGenerator** of the module **Parameters** in the top level design (see above).

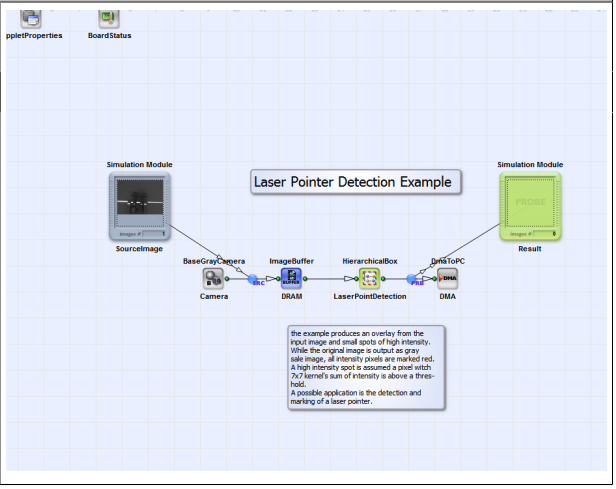
### 12.1.5.2. Design Versions

For JPEG color compression six example designs for microEnable 5 marathon platform for CameraLink and CoaxPress camera interfaces and for different link configurations and bandwidths are available. In each design an user library element for color JPEG compression, which is optimized for a specific platform and bandwidth is used. Please see the following table for the available design versions, the user library elements, the camera interface and the input bandwidth:

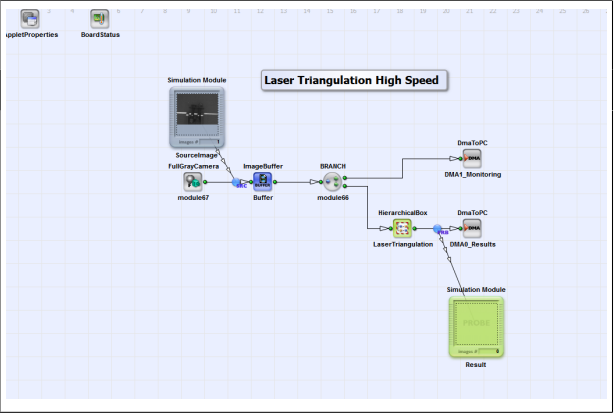
| Example | Camera Interface | User Library Element | Number of Processes | Input Bandwidth per Camera Interface |
|---|---|---|---|---|
| JPEG_DualBaseAreaBayer.va | Camera Link Base (mE5-MA-VCL) | **JPEG_Encoder_Color_300MPs_VCL** | 2 | Camera Link Base Speed 255MP/s: Parallelism 4, Design Clock Frequency: 125 MHz |
| JPEG_SingleFullAreaBayer.va | Camera Link Full (mE5-MA-VCL) | **JPEG_Encoder_Color_850MPs_VCL** | 1 | Camera Link Full Speed 850 MP/s: Parallelism 8, Design Clock Frequency: 170 MHz |
| JPEG_SingleCXP6x4AreaBayer.va | CoaxPress Quad Link (mE5-MA-VCX-QP) | **JPEG_Encoder_Color_2500MPs_VCX** | 1 | 2500 MPixel/s: Parallelism 16, Design Clock Frequency: 160 MHz |
| JPEG_DualCXP6x2AreaBayer.va | CoaxPress Dual Link (mE5-MA-VCX-QP) | **JPEG_Encoder_Color_800MPs_ VCX** | 1 | Mean Bandwidth 800 MPixel/s: Parallelism 8, Design Clock Frequency: 160 MHz |
| JPEG_TripleCXP6x1AreaBayer.va | CoaxPress Single Link (mE5-MA-VCX-QP) | **JPEG_Encoder_Color_600MPs_ VCX** | 1 | Mean Bandwidth 600 MPixel/s: Parallelism 4, Design Clock Frequency: 155 MHz |

Table 12.2. Design Versions for Color JPEG Encoding

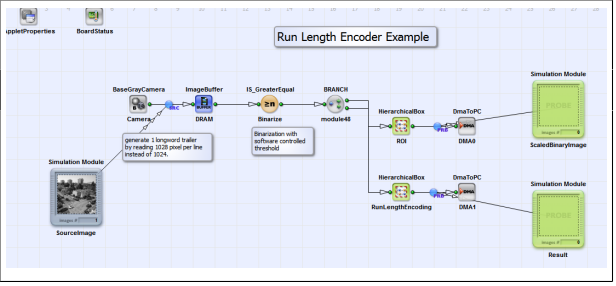
### 12.1.6. Laser Pointer Detection

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Advanced \LaserPointerDetection\LaserPointerDetection.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A convolution with high intensity spot coefficients is made. For results above threshold, the respective pixels are dyed in red. | |

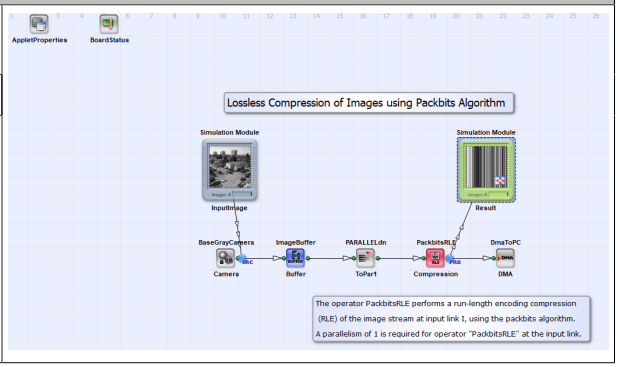
### 12.1.7. Laser Triangulation

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Advanced \LaserTriangulation\LaserTriangulation.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results. | |

### 12.1.8. Run Length Encoder

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Advanced \Compression\RunLengthEncoder.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A run length encoding example of defined format. Check the design comments for more information. | |

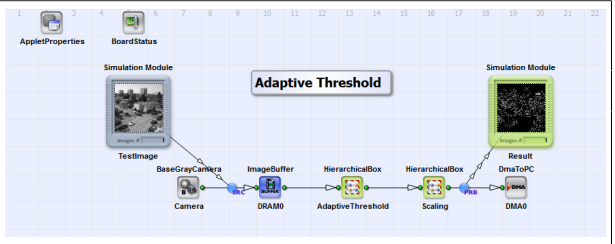
### 12.1.9. Packbits Run Length Encoder

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Compression \PackbitsRLE.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A packbits run length encoding example of defined format. Check the design comments for more information. | |

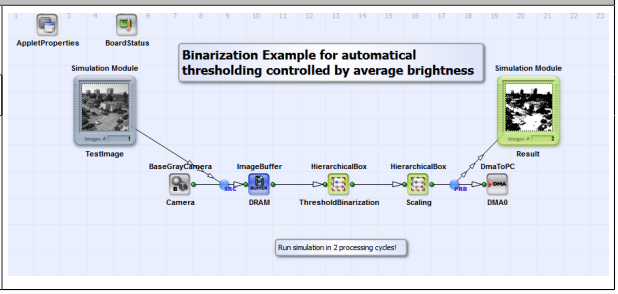
# 12.2. Binarization

In the following subsections you find four examples for binarization methods. A simple threshold binarization, an adaptive threshold algorithm, an example for automatical thresholding controlled by average brightness and a histogram threshold example are provided.

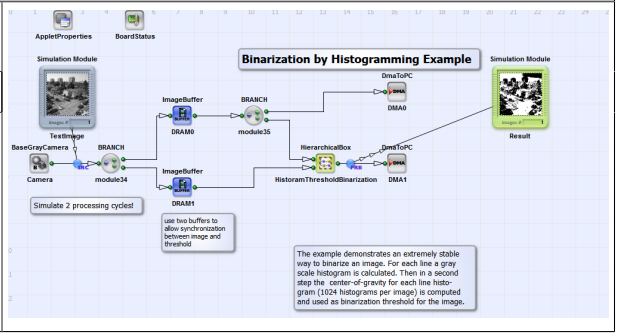
## 12.2.1. Adaptive Threshold

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Binarization \AdaptiveThreshold\AdaptiveThreshold.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A binarization example for local adaptive thresholding. A kernel size of 8 by 8 pixel is used. | |

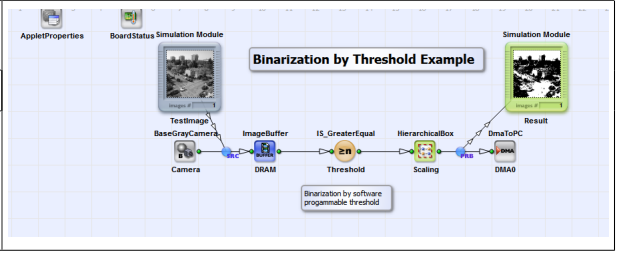
## 12.2.2. Auto Threshold Mean

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Binarization \AutoThresholdMean\AutoThresholdMean.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Determines the mean value of an image and used the value as threshold value for the next image processed. | |

## 12.2.3. Histogram Threshold

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Binarization \HistogramThreshold\HistogramThreshold.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Histogram thresholding. | |

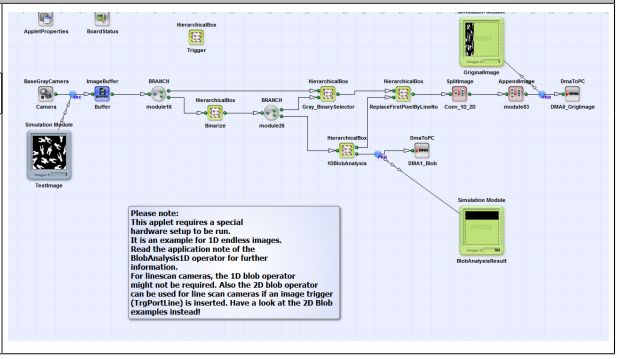
### 12.2.4. Simple Threshold Binarization

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Binarization \SimpleThreshold\SimpleThreshold.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Simple thresholding for binarization. | |

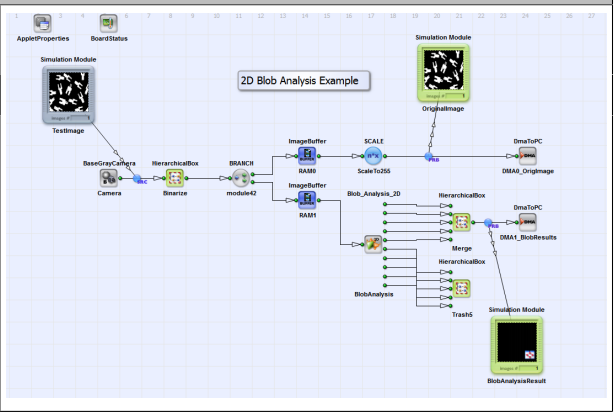
## 12.3. Blob Analysis

The example designs in this section are about the blob analysis of an image using the one and two dimensional operators **Blob_Analysis_1D** and **Blob_Analysis_2D**.

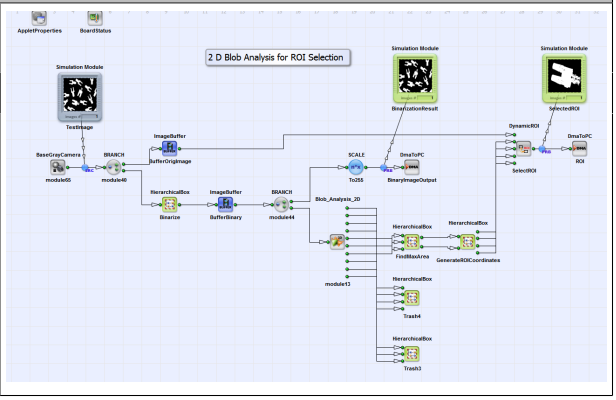
### 12.3.1. Blob 1D

| Brief Description | |
|---|---|
| **File:** \examples\Processing\BlobAnalysis\Blob1D \Blob1D.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows the usage of operator *Blob_Analysis_1D* in line scan applications. Please note that the example was build for a special hardware setup. | |

### 12.3.2. Blob 2D

| Brief Description | |
|---|---|
| **File:** \examples\Processing\BlobAnalysis\Blob2D\Blob2D.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels. |  |

### 12.3.3. Blob2D ROI Selection

| Brief Description | |
|---|---|
| **File:** \examples\Processing\BlobAnalysis\Blob2D_ROI_select\Blob2D_ROI_select.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image. |  |

## 12.4. Color

Find examples on color processing in the following sections. The examples contain color plane separation, Bayer demosaicing, RGB white balancing and Hue saturation intensity classification.

### 12.4.1. Bayer De-mosaicing

In literature different algorithms for interpolation of an color image from a Bayer Pattern image (Bayer demosaicing) exist. In this section we give several examples of such algorithms. In the following table they are summarized:

| Algorithm | Description | Example |
|---|---|---|
| **"Nearest Neighbor filter"** | according to [Ada95] | Section 12.4.1.1, 'Nearest Neighbor De-Mosaicing' |
| **"Bayer3x3filter"** | bilinear interpolation; based on VisualApplets operator **BAYER3x3Linear** | Section 12.4.1.2, 'Bayer 3x3 De-Mosaicing' |
| **"Bayer3x3filter"** with white balancing | bilinear interpolation; based on VisualApplets operator **BAYER3x3Linear** with additional white balancing | Section 12.4.1.4, 'Bayer 3x3 De-Mosacing with White Balancing' |

| Algorithm | Description | Example |
|---|---|---|
| **"Bayer5x5filter"** | bilinear interpolation; based on VisualApplets operator **BAYER5x5Linear** | Section 12.4.1.3, 'Bayer 5x5 De-Mosaicing' |
| **"Bayer5x5filter"** with white balancing | bilinear interpolation; based on VisualApplets operator **BAYER5x5Linear** with additional white balancing | Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing' |
| **"Laplace filter"** | edge sensitive | Section 12.4.1.6, 'Edge Sensitive Bayer De-Mosaicing Algorithm' |
| **"original Laroche filter"** | according to [Lar94] | Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche' |
| **"modified Laroche filter"** | according to [Lar94]; modified | Section 12.4.1.8, 'Modified Laroche Bayer De-Mosaicing Algorithm ' |
| **Bilinear Bayer Red/ BlueFollowedByGreen GreenFollowedByBlue/Red** | according to [Bas13] | Section 12.4.1.9, 'Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern Red/ BlueFollowedByGreen GreenFollowedByBlue/Red ' |
| **Bilinear Bayer Red/ BlueFollowedByBlue/Red GreenFollowedByGreen** | | Section 12.4.1.10, 'Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern RedFollowedByBlue GreenFollowedByGreen ' |
| **Bi-Color Bayer Pattern biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR** | | Section 12.4.1.11, 'Bayer DeMosaicing a Line Scan Camera with Bi-color Bayer Pattern biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR' |

Table 12.3. List of Bayer Demosaicing Examples

The algorithms named above are explained in detail in the corresponding sections. Here we give a short overview on the qualitative results obtained with the different demosaicing methods for an artificial test image (Figure 12.15, 'Artificial test image').



Figure 12.15. Artificial test image

The following figures show closer details on single structures. From left to right for: a) the **"original Laroche filter"** [Lar94], b) the **"modified Laroche filter"**, c) the **"Laplace filter"**, d) the **"Bayer5x5filter"**, e) the nearest neighbor interpolation method and for f) the **"Bayer3x3filter"**:

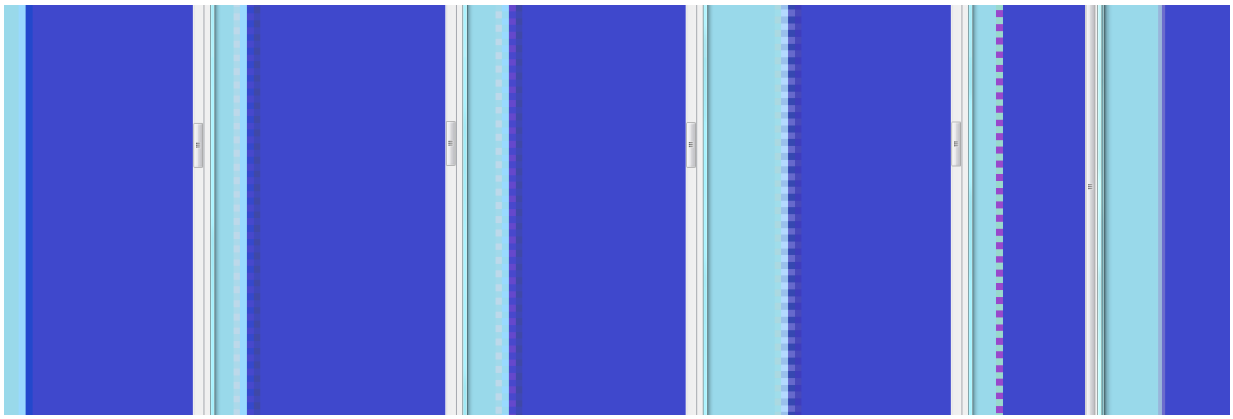


Figure 12.16. Straight edge

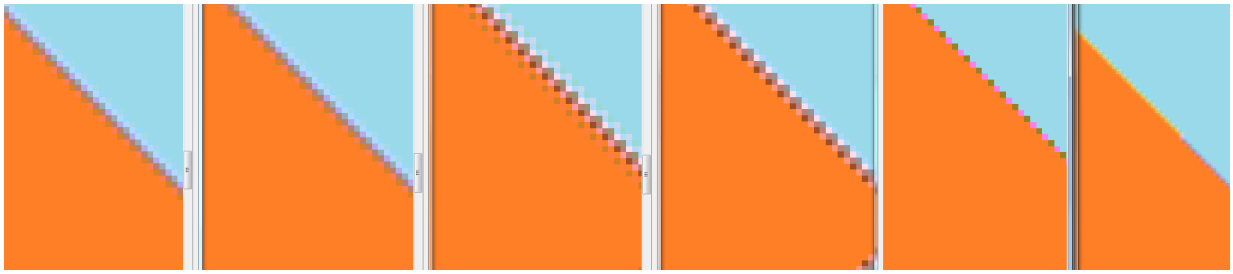


Figure 12.17. Diagonal edge

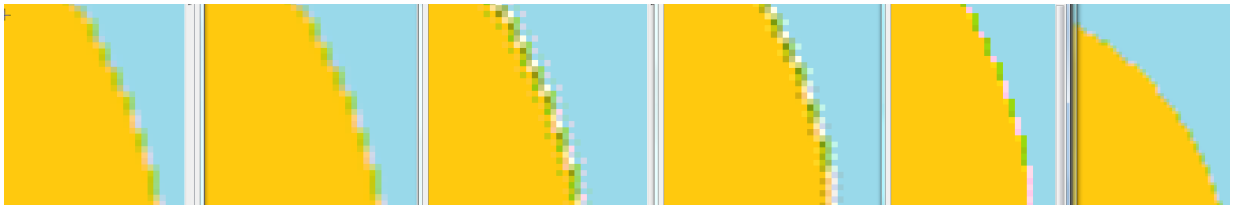


Figure 12.18. Curved edge

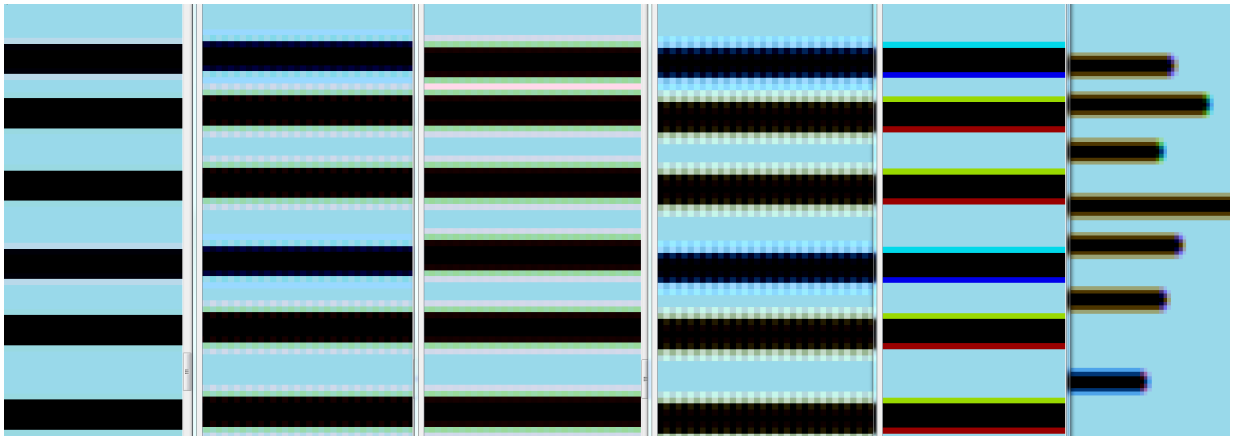


Figure 12.19. Periodic structure

For the artificial test image the **"original Laroche filter"** shows the best subjective result according to zipper- and false color effects. The **"modified Laroche filter"** has light zipper effects and is comparable to the **"Laplace filter"**. For the **"Laplace filter"** in addition many false color pixel appear at curved and diagonal edges. The **"Bayer5x5filter"** shows strong zipper effects, whereas the **"Nearest Neighbor filter"** shows strong false color effects. For straight edges strong zipper effects appear here. All edges appear blurry with the **"Bayer3x3filter"**.

**Attention**: The **"original Laroche filter"** is not always the best solution for Bayer demosaicing! It has no or very few zipper effects for an artificial test image but may have those for a noisy Bayer RAW image. During research we found, that the filtering algorithm is influenced by the noise.The **"modified Laroche" filter"** has zipper effects for the artificial test image but only light zipper effects for a noisy Bayer RAW image. Here the filtering algorithm is less than the **"original Laroche filter"** filter influenced by the noise. Also the **"Laplace filter"** has zipper effects for the artificial test image but no zipper effects for a noisy Bayer test image. False color effects appear here in addition. In a consequence you have to choose, which Bayer demosaicing algorithm is the best for your special purpose.

### 12.4.1.1. Nearest Neighbor De-Mosaicing

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \NearestNeighbor_maVCL.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Nearest Neighbor Bayer De-Mosaicing | |

#### 12.4.1.1.1. Theory

The nearest neighbor interpolation method is the simplest Bayer demosaicing algorithm. For a Bayer pattern [Bay76] of four neighboring pixels P1 to P4 (see Figure 12.20, 'Bayer pattern') the missing red (R), green (G) and blue (B) colors can be interpolated according to [Ada95]:

$$\begin{aligned} R2 = R3 = R4 = R1, \\ B1 = B2 = B3 = B4, \\ G1 = G2, \\ G3 = G4\ . \end{aligned} \tag{12.1}$$

Here a pattern Red-followed-by-Green is shown, but this interpolation method is equivalent for all Bayer patterns.

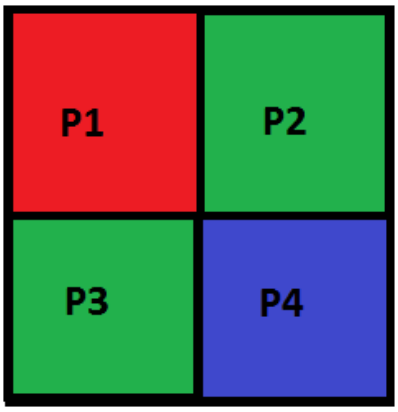



Figure 12.20. Bayer pattern

### 12.4.1.1.2. VisualApplets Design

In VisualApplets an example design for nearest neighbor interpolation is implemented. You can find it under \examples\Processing\Color\Bayer\NearestNeighbor_maVCL.va. You can see its basic design structure in Figure 12.21, 'Basic design structure'. The content of the HierarchicalBoxes **InputImage** and **OutputImage** is equivalent to a basic acquisition design (see e.g. Section 10.3, 'Basic Acquisition Examples for Camera Link Cameras for marathon, LightBridge and ironman Frame Grabbers').

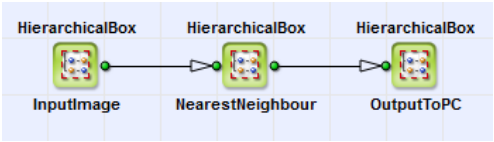



Figure 12.21. Basic design structure

In Figure 12.22, 'Content of **NearestNeighbour**' you can see the content of the HierarchicalBox **NearestNeighbour**. A $2 \times 2$ kernel is defined. The current pixel position is here the upper left corner. The colors red, green and blue are calculated according to Equation 12.1 in the HierarchicalBoxes **Red**, **Green**, **Blue** and with **CurrentPixel** using the VisualApplets operator **SelectSubkernel**. In **WhereAmI_Colour** the color of the current pixel position in the Bayer pattern is determined using Modulo-,AND-, NOT and NotEqual-operators. The comment boxes in the design give detailed information of the color at the current position and at every link. With three **IF** operators the missing colors are added to the current pixel. All three colors are merged with the operator **MergeComponents**.

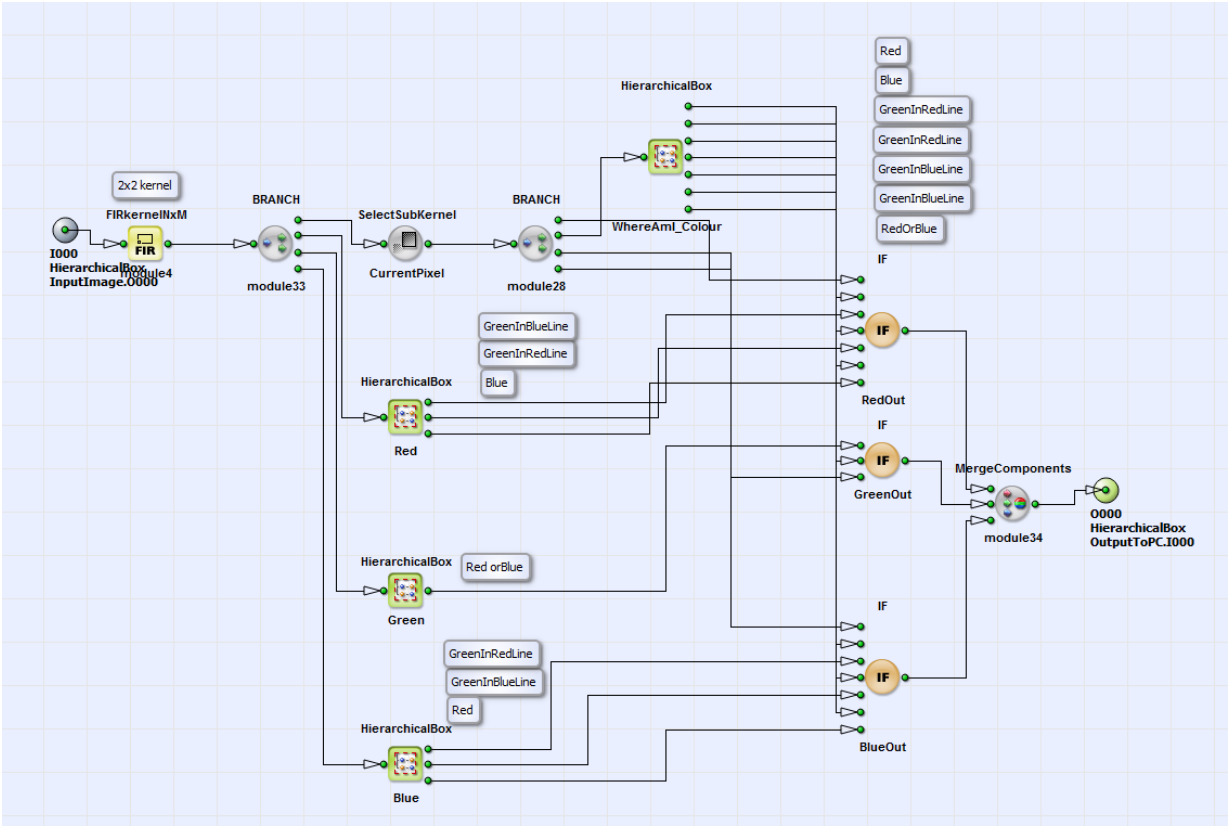



Figure 12.22. Content of **NearestNeighbour**

### 12.4.1.2. Bayer 3x3 De-Mosaicing

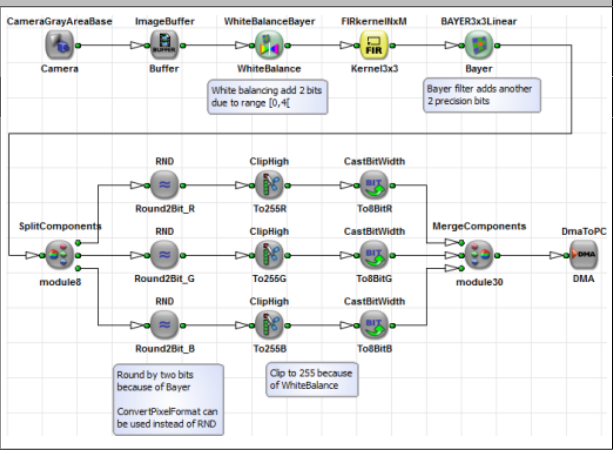
| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \Bayer3x3.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Note the usage of operator *ConvertPixelFormat* to reduce the bit width to 8 bit per component. | |



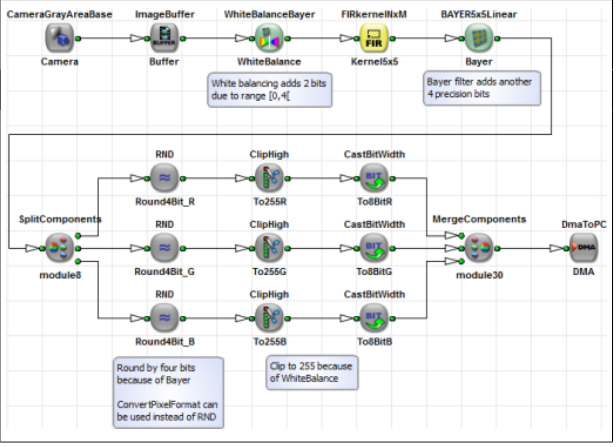
### 12.4.1.3. Bayer 5x5 De-Mosaicing

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \Bayer3x3.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Note the usage of operator *ConvertPixelFormat* to reduce the bit width to 8 bit per component. | |

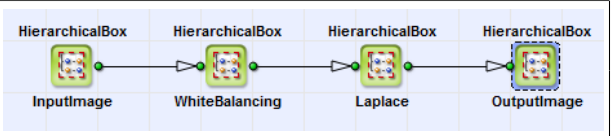
### 12.4.1.4. Bayer 3x3 De-Mosacing with White Balancing

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \BayerWhiteBalancing3x3.va |  |
| **Default Platform:** mE4VD4-CL | |
| **Short Description**<br><br>The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Moreover, a white balancing for color correction is added. | |

### 12.4.1.5. Bayer 5x5 De-Mosacing with White Balancing

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \BayerWhiteBalancing5x5.va |  |
| **Default Platform:** mE4VD4-CL | |
| **Short Description**<br><br>The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Moreover, a white balancing for color correction is added. | |

### 12.4.1.6. Edge Sensitive Bayer De-Mosaicing Algorithm

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \LaplaceFilter_maVCL.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Laplace Edge Sensitive Bayer De-Mosaicing | |

In this section we give an example of an edge sensitive Bayer demosaicing algorithm, which we call **"Laplace filter"**. In the following we first describe the interpolation algorithm and introduce shortly the implementation in VisualApplets afterwards.

#### 12.4.1.6.1. Interpolation Algorithm

For the interpolation of the green values on red and blue pixels gradients in vertical and horizontal directions ($\Delta h$ and $\Delta v$) are used:

$$\begin{aligned}\Delta v &= |G_{x,y-1} - G_{x,y+1}| + |2 \cdot I_{x,y} - I_{x,y-2} - I_{x,y+2}|, \\ \Delta h &= |G_{x-1,y} - G_{x+1,y}| + |2 \cdot I_{x,x} - I_{x-2,y} - I_{x+2,y}| \ .\end{aligned} \tag{12.2}$$

$I_{x,y}$ is the chrominance value (red or blue) on the pixel where the green value has to be interpolated. Green $G_{x,y}$ is then on the red and blue pixel positions:

$$\begin{aligned}&\text{If } \Delta v < \Delta h - c: \\ G_{x,y} = A &= \frac{2 \cdot [G_{x,y-1} + G_{x,y+1} + I_{x,y}] - I_{x,y-2} - I_{x,y+2}}{4}, \\ &\text{If } \Delta v - c > \Delta h: \\ G_{x,y} = B &= \frac{2 \cdot [G_{x-1,y} + G_{x+1,y} + I_{x,y}] - I_{x-2,y} - I_{x+2,y}}{4}, \\ &\text{If } |\Delta v - \Delta h| \leq c: \\ G_{x,y} &= \frac{A + B}{2} \ .\end{aligned} \tag{12.3}$$

Here the value $c$ determines the significance of an edge. For the interpolation of the colors red and blue there exist three cases:

1. Interpolation of color $I_{x,y}$ on a green pixel in a line with the same color:

$$I_{x,y} = \frac{2 \cdot [I_{x-1,y} + I_{x+1,y} + G_{x,y}] - G_{x-2,y} - G_{x+2,y}}{4} \ . \tag{12.4}$$

2. Interpolation of color $I_{x,y}$ on a green pixel in a line with other color:

$$I_{x,y} = \frac{2 \cdot [I_{x,y+1} + I_{x,y-1} + G_{x,y}] - G_{x,y+2} - G_{x,y-2}}{4} \ . \tag{12.5}$$

3. For the interpolation of a color on a pixel with other color gradients $\Delta d1$ and $\Delta d2$ are used:

$$\begin{aligned}\Delta d1 &= |I_{x-1,y+1} - I_{x+1,y-1}| + |2 \cdot I_{x,y} - I_{x+2,y-2} - I_{x-2,y+2}| \ , \\ \Delta d2 &= |I_{x-1,y-1} - I_{x+1,y+1}| + |2 \cdot I_{x,y} - I_{x-2,y-2} - I_{x+2,y+2}| \ .\end{aligned} \tag{12.6}$$

The color (red or blue) is then:

$$\begin{aligned}&\text{If } \Delta d1 < \Delta d2 - c: \\ I_{x,y} = C &= \frac{2 \cdot [I_{x-1,y-1} + I_{x+1,y+1} + I_{x,y}] - I_{x+2,y+2} - I_{x-2,y-2}}{4}, \\ &\text{If } \Delta d1 - c > \Delta h: \\ I_{x,y} = D &= \frac{2 \cdot [I_{x+1,y-1} + I_{x-1,y+1} + I_{x,y}] - I_{x+2,y-2} - I_{x-2,y+2}}{4}, \\ &\text{If } |\Delta d1 - \Delta d2| \leq c: \\ G_{x,y} &= \frac{C + D}{2} \ .\end{aligned} \tag{12.7}$$

## 12.4.1.6.2. Implementation in VisualApplets

You can find the VisualAplets design for the **Laplace filter"** under \examples\Processing\Color\Bayer \LaplaceFilter_maVCL.va. In Fig. 12.23 you can see the basic design structure. The content of **InputImage** and **OutputImage** is equivalent to the basic acquisition examples (see e.g. Section 10.3, 'Basic Acquisition Examples for Camera Link Cameras for marathon, LightBridge and ironman Frame Grabbers').

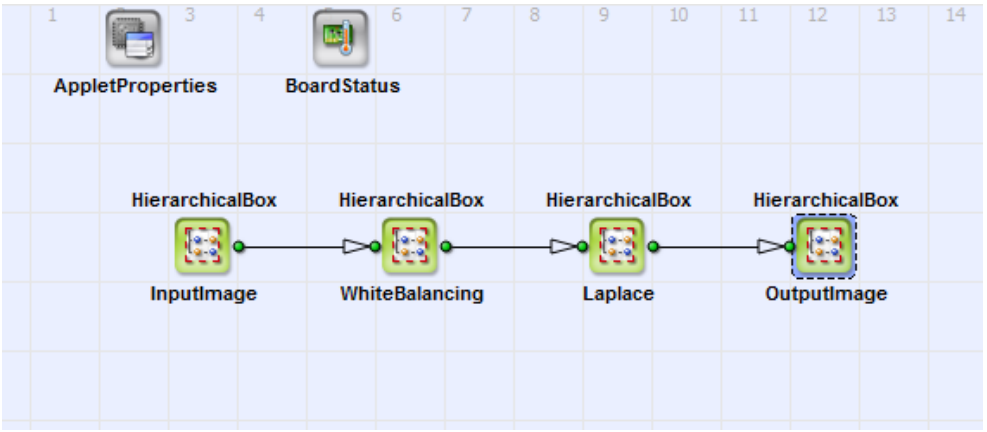



Figure 12.23. Basic design structure

In Figure 12.24, 'Content of HierarchicalBox **Laplace**' the content of the HierarchicalBox **Laplace** is shown.

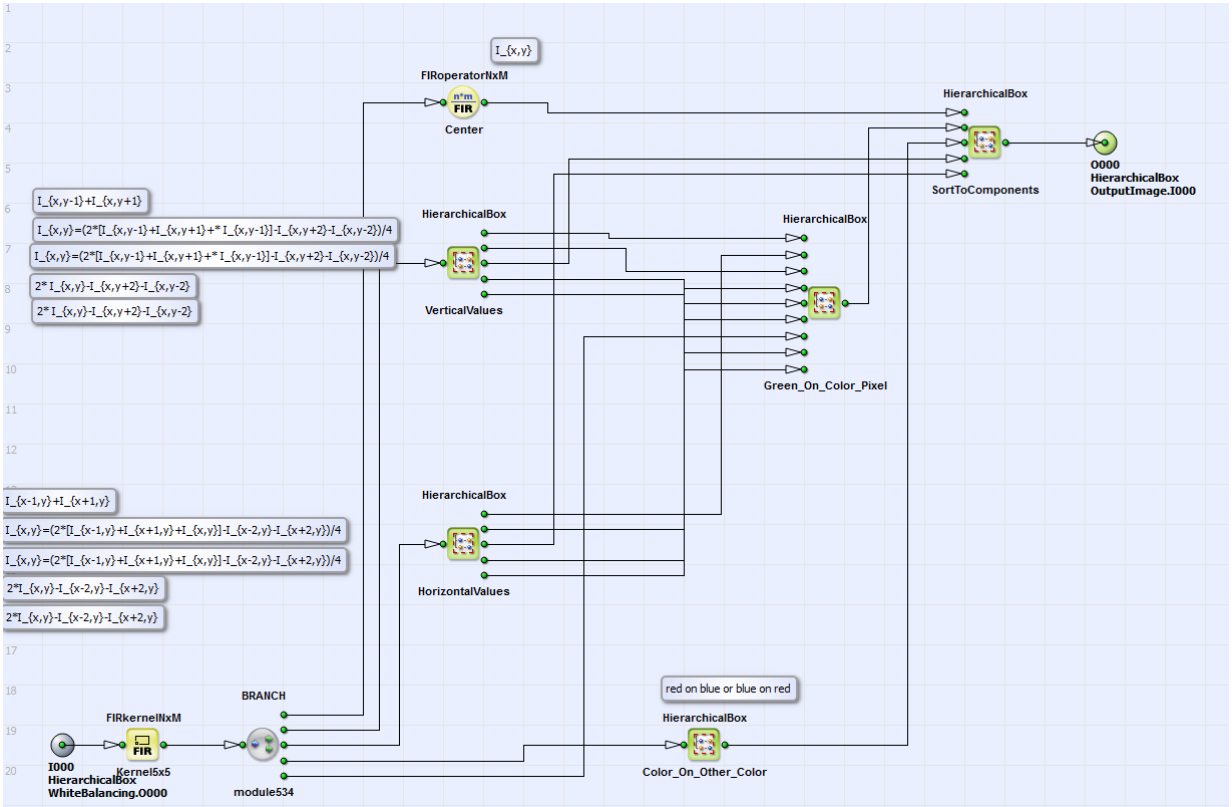



Figure 12.24. Content of HierarchicalBox **Laplace**

In the HierarchicalBoxes **HorizontalValues**, **VerticalValues**, **Green_On_Color_Pixel** and **Color_On_Other_Color** the single kernel components of a $5 \times 5$ kernel are used for calculation of the single equation steps of Equation 12.2 to Equation 12.7. Comment boxes give detailed information on the content of each HierarchiaclBox and the current equation step.

In Figure 12.25, 'Content of HierarchicalBox **SortToComponents**' you can see the content of the box **SortToComponents**. Here the calculated color values red, green and blue are assigned to the output value in dependence on the current pixel color. The color of the current pixel is determined in the HierarchicalBox **WhereAmI_Colour**. Here different you can choose different Bayer patterns as explained in the corresponding comment box in the example.

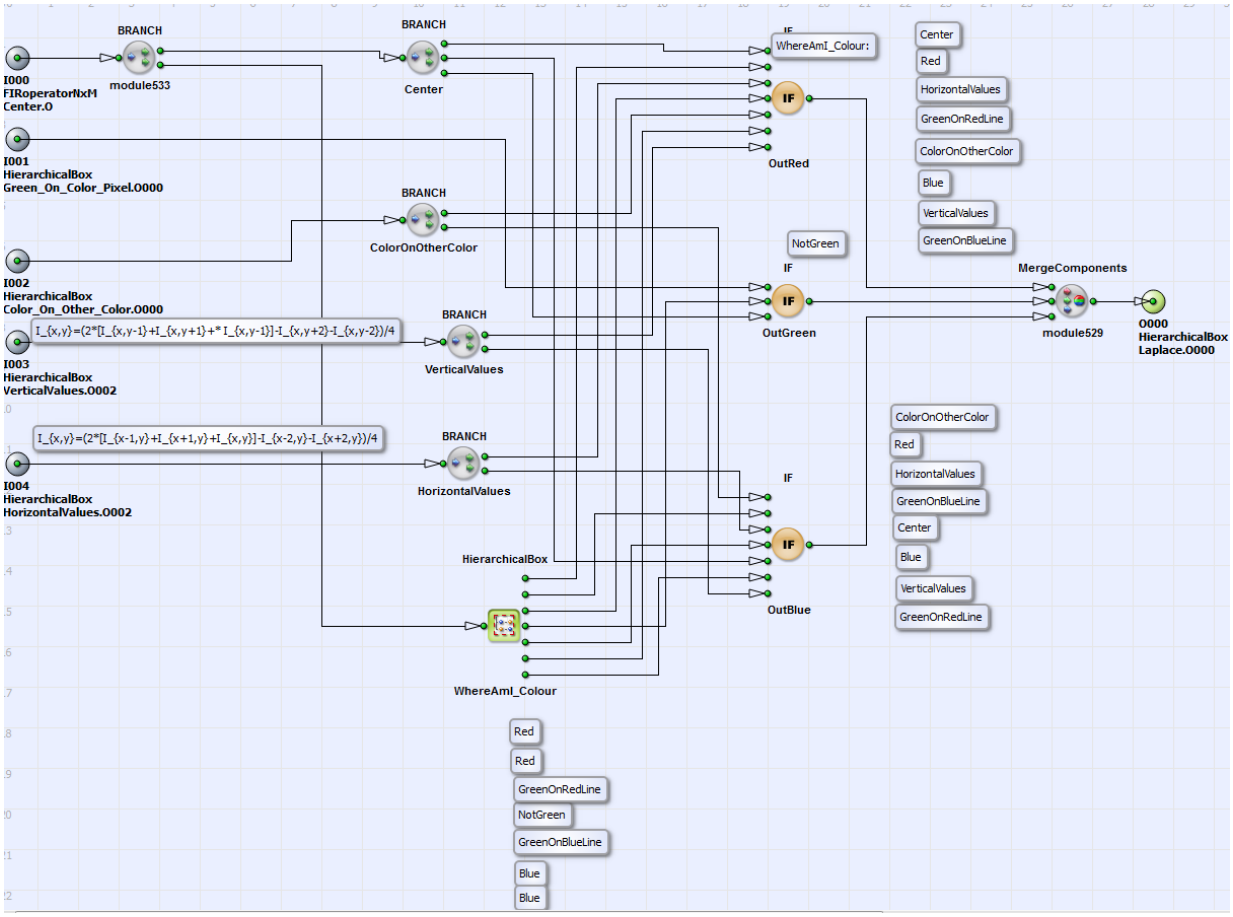
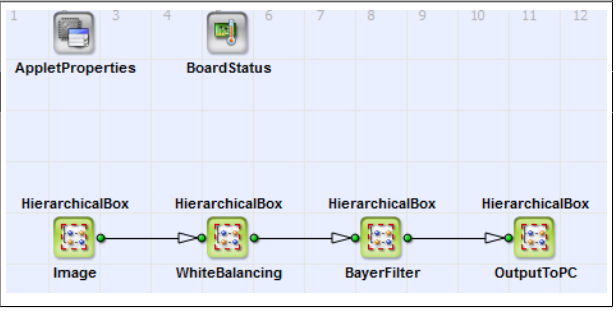



Figure 12.25. Content of HierarchicalBox **SortToComponents**

## 12.4.1.7. Bayer De-Mosaicing Algorithm According to Laroche

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \Laroche_original_maVCL.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Bayer De-Mosaicing Algorithm According to Laroche | |



According to Laroche et al. [Lar94] the red, green and blue pixel values from a Bayer-Pattern [Bay76] can be extrapolated using the following algorithm:

### 12.4.1.7.1. Interpolation Step 1

In a first step all green pixel values (luminance values) are extrapolated with horizontal and vertical chrominance gradients ($\Delta H$ and $\Delta V$). The gradients are defined using a $5 \times 5$ kernel:

$$\begin{aligned} \Delta H &= |\frac{I_{x-2,y} + I_{x+2,y}}{2} - I_{x,y}| \ , \\ \Delta V &= |\frac{I_{x,y-2} + I_{x,y+2}}{2} - I_{x,y}| \ . \end{aligned} \tag{12.8}$$

$I_{x,y}$ is the chrominance value (red or blue) on the pixel where the green value has to be interpolated. Green $G_{x,y}$ is then on the red and blue pixel positions:

$$\begin{aligned} &\text{If } \Delta H < \Delta V: \\ G_{x,y} &= \frac{G_{x-1,y} + G_{x+1,y}}{2} \ , \\ &\text{If } \Delta V < \Delta H: \\ G_{x,y} &= \frac{G_{x,y-1} + G_{x,y+1}}{2} \ , \\ &\text{If } \Delta H \approx \Delta V: \\ G_{x,y} &= \frac{G_{x-1,y} + G_{x+1,y} G_{x,y-1} + G_{x,y+1}}{4} \ . \end{aligned} \tag{12.9}$$

### 12.4.1.7.2. Interpolation Step 2

From the interpolated luminance the missing red and blue pixel values can be calculated [Lar94]. Blue on a red pixel or red on a blue pixel:

$$I_{x,y} = \frac{(I_{x-1,y-1} - G_{x-1,y-1}) + (I_{x+1,y-1} - G_{x+1,y-1}) + (I_{x+1,y+1} + G_{x+1,y+1}) + (I_{x-1,y+1} - G_{x-1,y+1})}{4} + G_{x,y} \ , \tag{12.10}$$

blue on a green pixel in a red line or red on a green pixel in a blue line:

$$I_{x,y} = \frac{(I_{x,y-1} - G_{x,y-1}) + (I_{x,y+1} - G_{x,y+1})}{2} + G_{x,y} \ , \tag{12.11}$$

and blue on a green pixel in a blue line and red on a green pixel in a red line:

$$I_{x,y} = \frac{(I_{x-1,y} - G_{x-1,y}) + (I_{x+1,y} - G_{x+1,y})}{2} + G_{x,y} \ . \tag{12.12}$$

Here $I_{x,y}$ is the color value red or blue, that has to be determined. The red, green and blue values on the red, green and blue pixels are the ones from the Bayer-Pattern input image.

### 12.4.1.7.3. VisualApplets-Design

The complete algorithm (\examples\Processing\Color\Bayer\Laroche_original_maVCL.va) is implemented in a VisualApplets design for a microEnable5 frame grabber (marathon, Lightbride or ironman). You can see its basic design structure in Figure 12.26, 'Basic design structure'. The content of the HierarchicalBoxes **Image** and **OutputToPC** is equivalent to the basic acquisition designs, see e.g. Section 10.3, 'Basic Acquisition Examples for Camera Link Cameras for marathon, LightBridge and ironman Frame Grabbers'. In the HierarchicalBox **WhiteBalancing** a white balancing equivalent to the example in Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing' is performed. In Figure 12.27, 'Interpolation step 1 of the Bayer-demosaicing process' to Figure 12.29, 'Content of the HierarchicalBox **BlueAndRed**' the content of box **BayerFilter** with detailed comments is shown: In Figure 12.27, 'Interpolation step 1 of the Bayer-demosaicing process' the gradients $\Delta H$ and $\Delta V$ are calculated according to Equation 12.8. The green values on red and blue pixels are determined according to Equation 12.9.

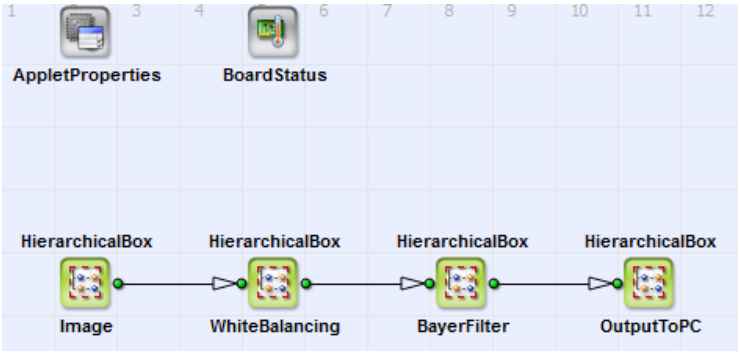



Figure 12.26. Basic design structure

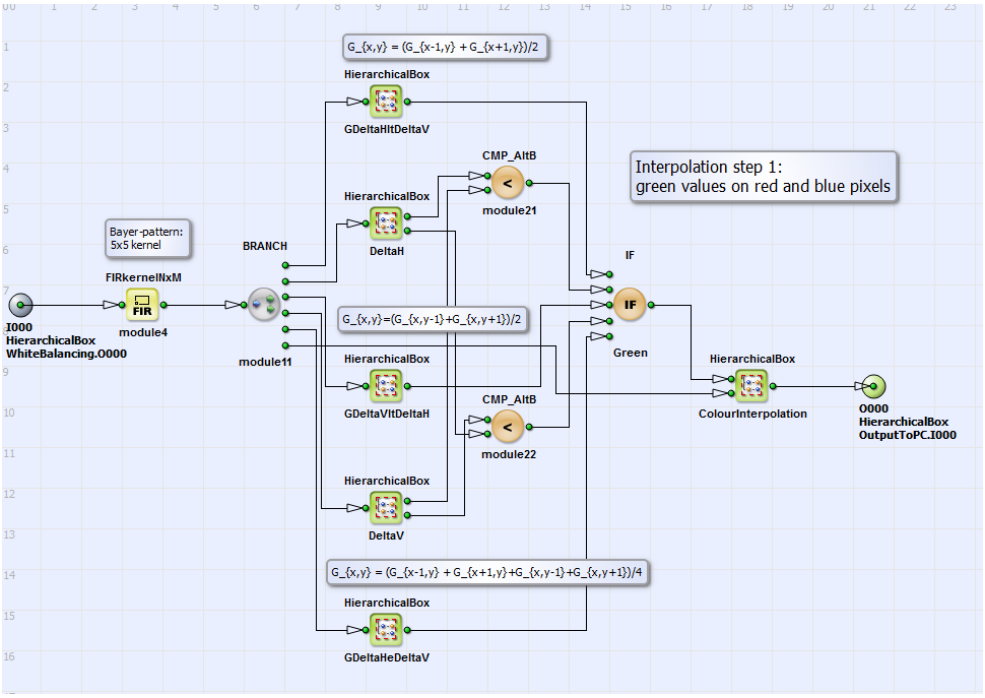



Figure 12.27. Interpolation step 1 of the Bayer-demosaicing process

In Figure 12.28, 'Content of **ColourInterpolation**' the content of the HierarchicalBox **ColourInterpolation** is displayed: In the HierarchicalBox **GreenOut** the green values of interpolation step 1 are determined as output green values if the current position is a red or blue pixel. Otherwise, if the current position is a green pixel, the output green value is the input green value from the Bayer-pattern. The position determination takes place in **WhereAmI RedOrBlue** using Modulo-,AND-, NOT and NotEqual-operators. In the HierarchicalBox **BlueAndRed** all red and blue values are interpolated from the input Bayer-pattern (using a $3 \times 3$ kernel) and the interpolated green values according to Equation 12.9. In Figure 12.29, 'Content of the HierarchicalBox **BlueAndRed**' you can see the content of the hierarchical box **BlueAndRed** with the interpolation step 2. The synchronization is performed not for a kernel but for a sum of kernel values to save resources on the FPGA. Finally (see Figure 12.28, 'Content of **ColourInterpolation**') with the IF-operators **RedOut** and **BlueOut** the calculated red and blue values are assigned to the corresponding pixel positions, which are determined in the HierarchicalBox **WhereAmIColour** (analog to **WhereAmI RedOrBlue**).

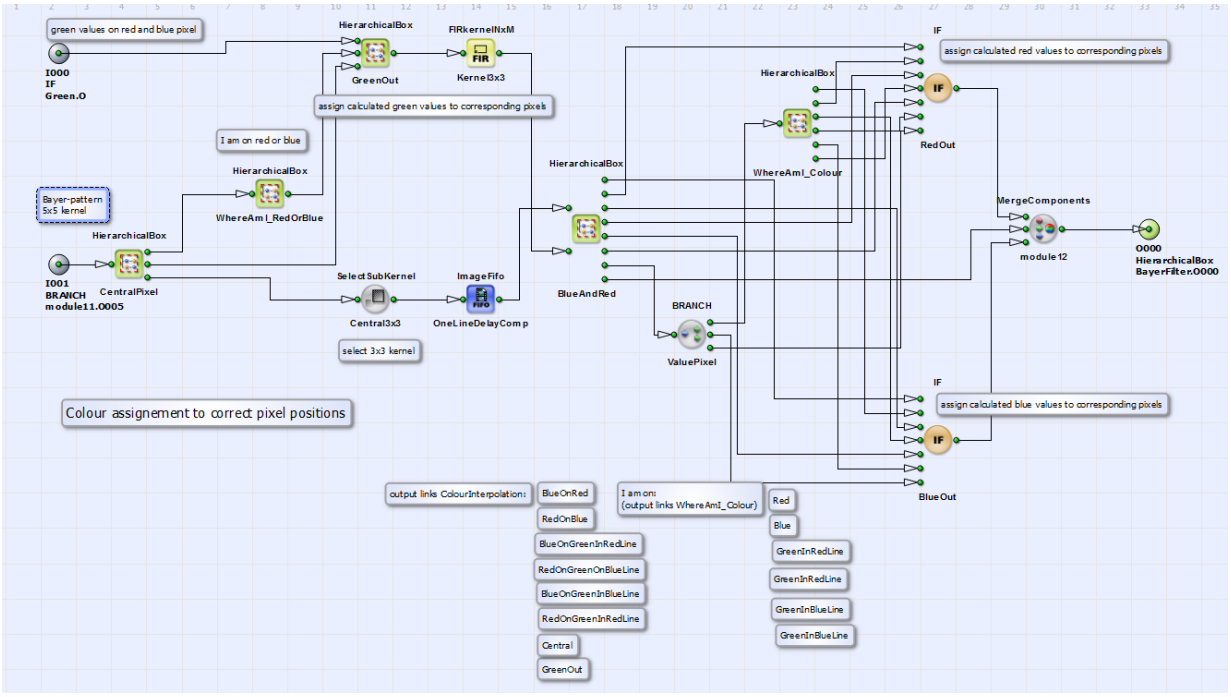



Figure 12.28. Content of **ColourInterpolation**

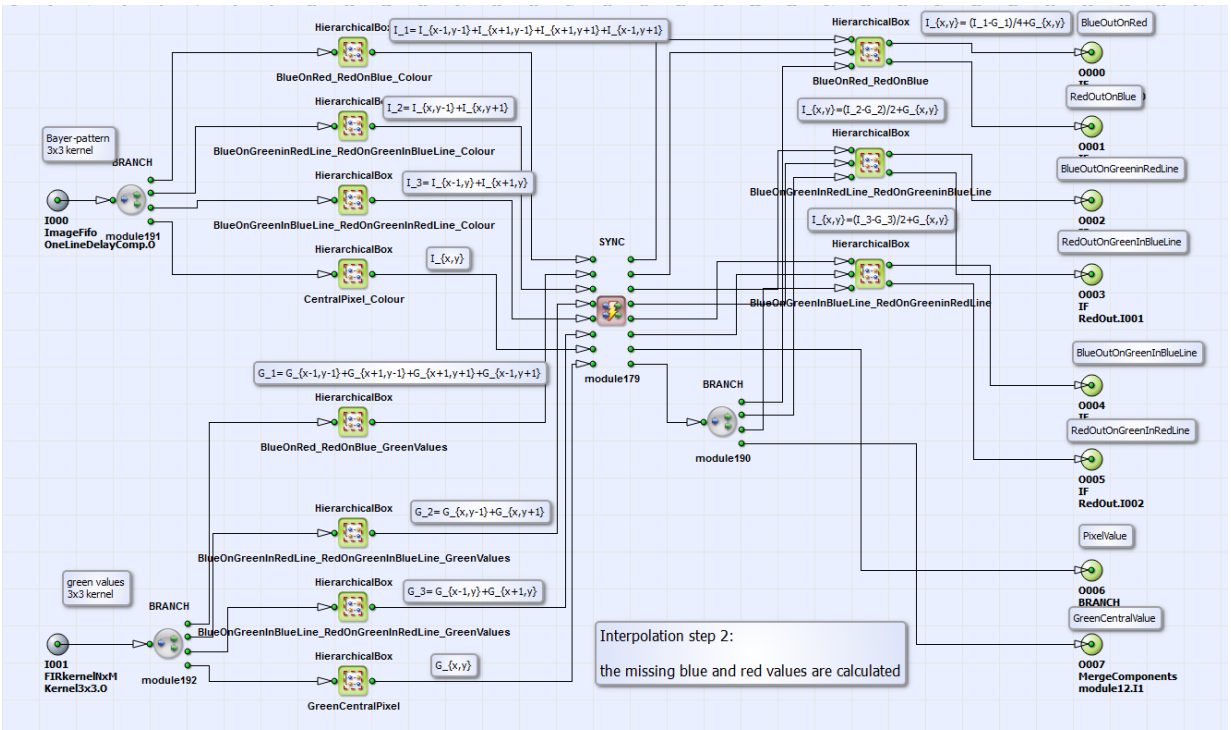



Figure 12.29. Content of the HierarchicalBox **BlueAndRed**

## 12.4.1.7.4. Result of the Bayer-Demosaicing Process

In Figure 12.31, 'Image demosaiced with the algorithm of Laroche et al. [Lar94]' you can see the result of the Bayer-demosaicing process with the implemented algorithm in comparison to the original artificial color image (Figure 12.30, 'Original color image') and an image demosaiced with an bilinear $5 \times 5$ interpolation algorithm, using a VA-standard operator **Bayer5x5** (Figure 12.32, 'Image demosaiced with an bilinear $5 \times 5$ algorithm').



Figure 12.30. Original color image



Figure 12.31. Image demosaiced with the algorithm of Laroche et al. [Lar94]

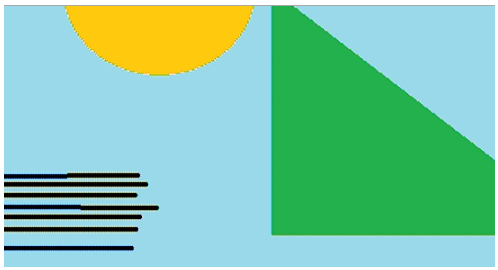
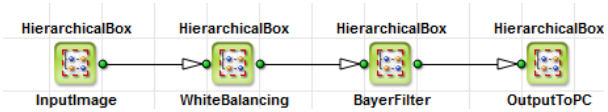


Figure 12.32. Image demosaiced with an bilinear $5 \times 5$ algorithm

As a result you can see a reduced zipper effect and reduced false color artefacts in comparison to the $5 \times 5$ interpolation method.

### 12.4.1.8. Modified Laroche Bayer De-Mosaicing Algorithm

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \Laroche_modified_maVCL.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Resource optimized Bayer De-Mosaicing Algorithm According to Laroche | |

In this section we describe a slightly modified version of the Bayer demosaicing algorithm of Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche'. The only difference in the interpolation algorithm (see Equation 12.8 to Equation 12.12) is, that the green color values ($G_{x-1,y-1}$, $G_{x+1,y+1}$, $G_{x,y-1}$, $G_{x,y+1}$, $G_{x,y-1}$, $G_{x,y+1}$) for the interpolation of the red and blue color values (Equation 12.10 to Equation 12.12) are bilinearly interpolated. All other color interpolation steps are equivalent to the **"original Laroche filter"**. In Section 12.4.1, 'Bayer De-mosaicing ' the qualitative difference in the Bayer demosaicing results are presented. You can find the

corresponding VisualApplets example for the modified version under \examples\Processing\Color \Bayer\Laroche_modified_maVCL.va. It is implemented for a microEnable 5 frame grabber for 8 bit input bit depth and a parallelism of 2 but can easily adapted to a version for a microEnable IV frame grabber, other input bit depths or parallelisms. In the following shortly the modified parts in the VisualApplets design are presented. In Figure 12.33, 'Content of **ColourInterpolation** for the modified Laroche filter.' you can see the content of the HierarchicalBox **ColourInterpolation** (see Figure 12.27, 'Interpolation step 1 of the Bayer-demosaicing process').

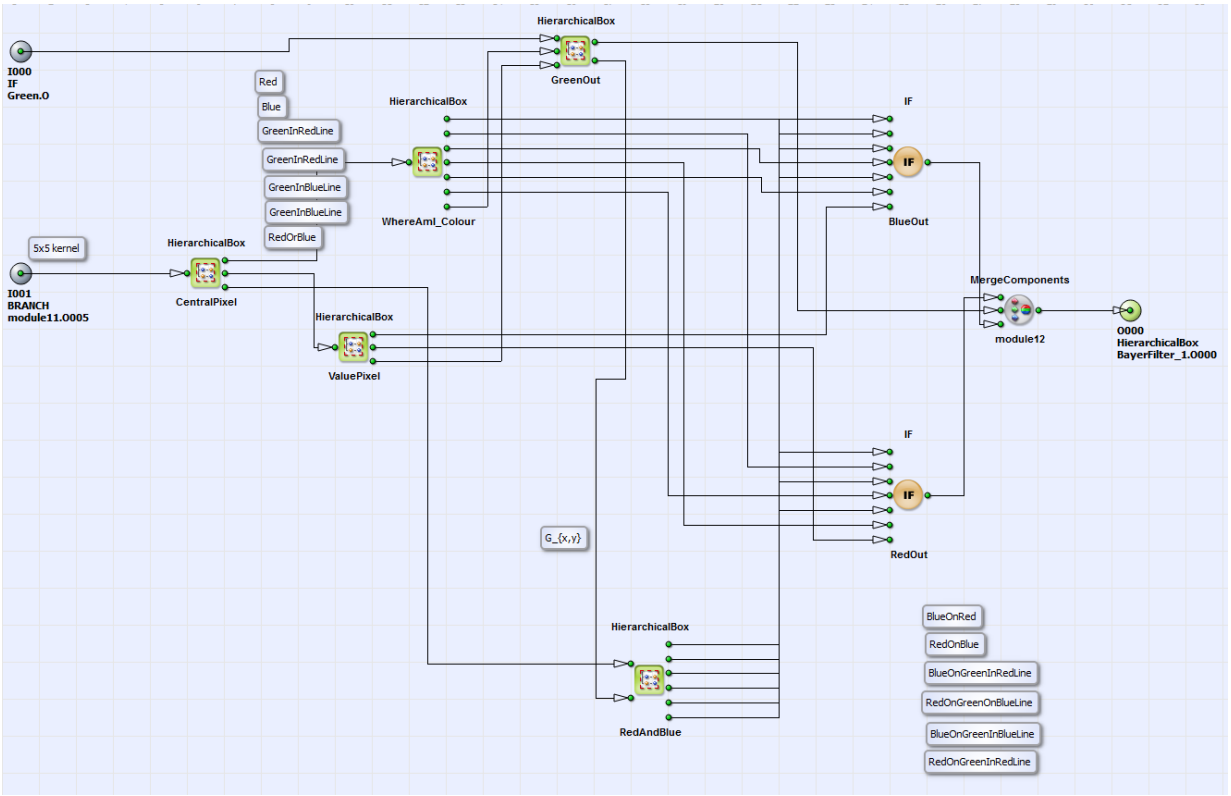
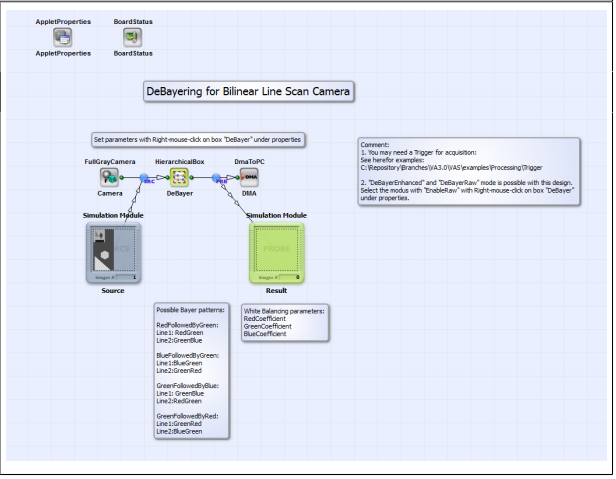



Figure 12.33. Content of **ColourInterpolation** for the modified Laroche filter.

The red and blue color values are interpolated in the box **RedandBlue** out of a $5 \times 5$) kernel of a Bayer pattern and the calculated green value $G_{x,y}$ at the current pixel position. The calculation of the green value is done in the HierarchicalBox **GreenOut**. Equivalent to the example in Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche' the color at the current pixel position is found in **whereAmI_Colour**. All three color components are finally merged together.

## 12.4.1.9. Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern Red/BlueFollowedByGreen GreenFollowedByBlue/Red

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \BilinearBayer_RG_GB.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>The example shows the de-mosaicing of a Bayer RAW pattern of a bilinear line scan camera with color pattern Red/ BlueFollowedByGreen_GreenFollowedByBlue/ Red in "Bayer Enhanced Raw" and "Bayer Raw" mode. | |



In the Visual Applets design example "BilinearBayer_RG_GB.va" the Bayer demosaicing for a bilinear line scan camera with colors red or blue followed by green in the first sensor (top) line and colors green followed by blue or red in the second (bottom) sensor line (or vice versa) is implemented. See Fig. 12.34 for the visualization of the sensor layout.

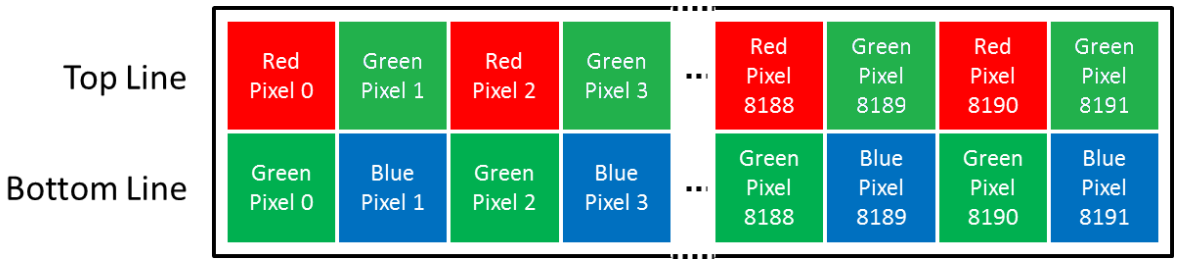



Figure 12.34. Sensor layout of a bilinear line scan camera with color pattern Red/ BlueFollowedByGreen_GreenFollowedByBlue/Red

The Bayer demosaicing for this sensor layout is implemented in the applet for two camera modes: the "Bayer Raw" mode and the "Bayer Enhanced Raw" mode. In the first case the motion between two acquisitions is two sensor lines, in the second case one line. Due to this, one color component is acquired for each pixel in the "Bayer Raw" mode and two color components (red and green or green and blue) per pixel in the "Bayer Enhanced Raw" mode. Please find further informations on the camera modes for example under [Bas13]. In Fig. 12.35 you can see the basic structure of the design "BilinearBayer_RG_GB.va".

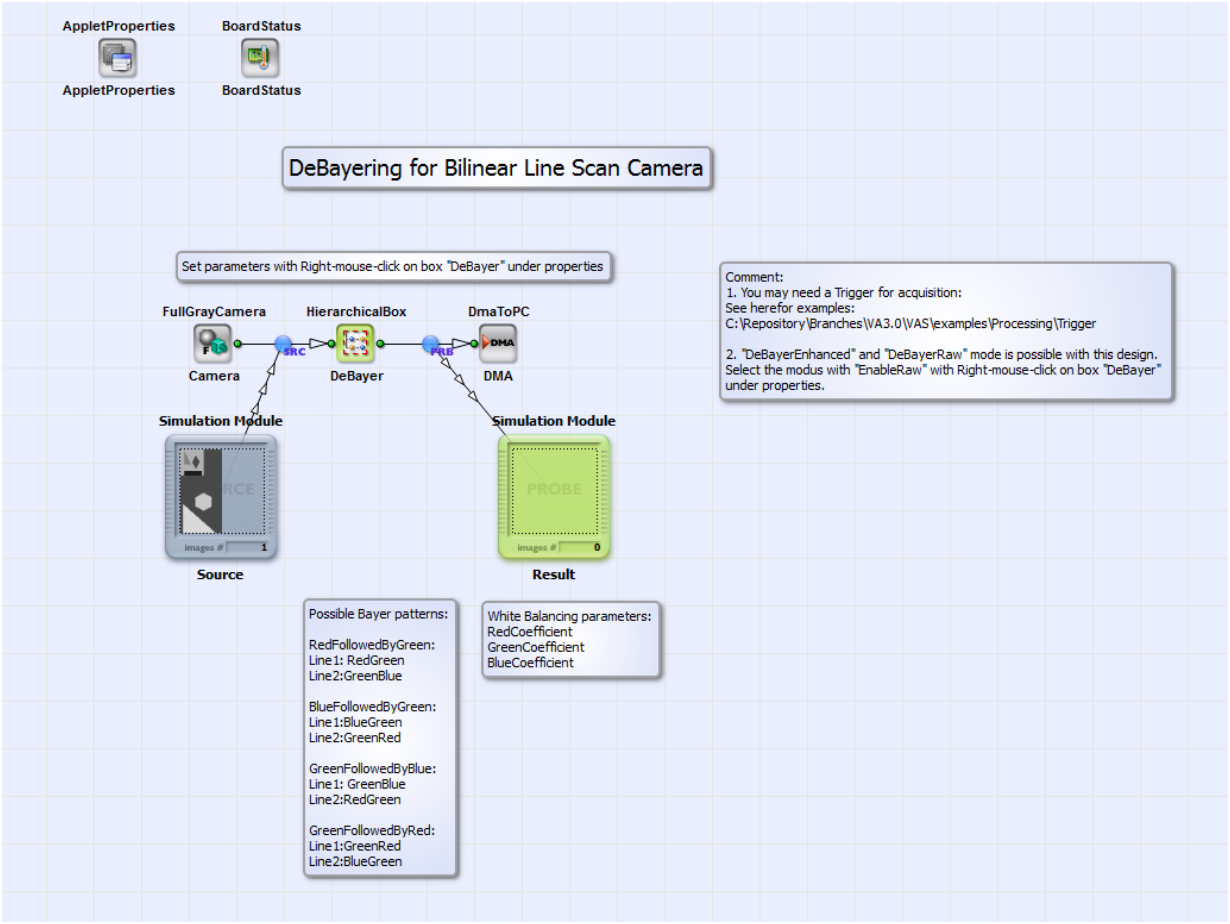



Figure 12.35. Basic design structure of "BilinearBayer_RG_GB.va"

An image acquired in "Bayer Raw" or "Bayer Enhanced Raw" mode from an bilinear line scan camera in Camera Link Full configuration (see camera interface operator **FullGrayCamera**) is demosaiced in the HierarchicalBox **DeBayer**. With "right-mouse-click" on this box you can choose the Bayer pattern under "Properties" and the white balancing coefficients for red, green and blue. The fully color interpolated rgb image is transmitted to PC via DMA. The default platform for this design is microEnable 5 marathon VCL, but it can easily be adapted to another platform. For the design you can find test images for the "Bayer Raw" mode and the "Bayer Enhanced Raw" mode under \examples\Processing\Color\Bayer \TestImagesBilinearBayer. The content of box **DeBayer** is shown in Fig. 12.36.

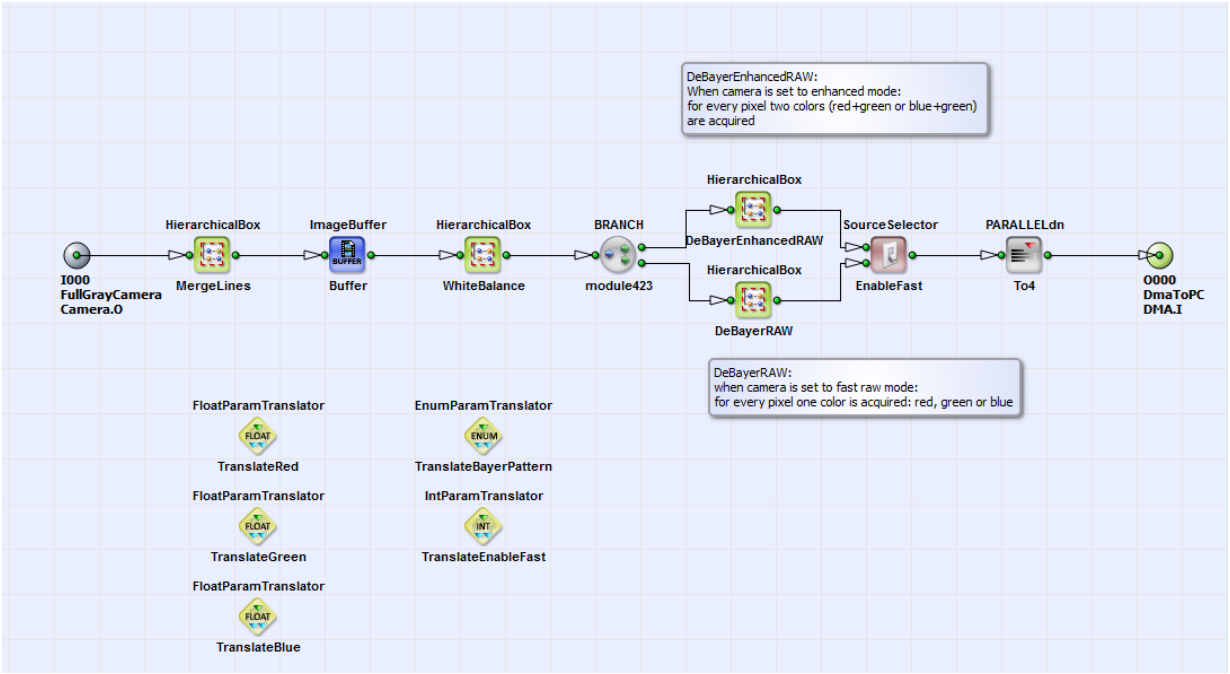



Figure 12.36. Content of HierarchicalBox **DeBayer**

In the HierarchicalBox **MergeLines** the current 8 bit pixel value is merged with its neighbor in the next line to one 16 bit value. In the HierarchicalBoax **WhiteBalance** a white balancing of the red, green and blue components is performed. You can choose the white balancing coefficients directly with "double-mouse-click" on the operator **WhiteBalanceBayer** in this box or with "right-mouse-click" on the box "DeBayer" as described above. With the operator **SourceSelector_EnableFast** you can choose the DeBayer mode **DeBayerEnhancedRAW** or **DeBayerRAW**. In Fig. 12.37 you can see the content of box **DeBayerEnhancedRAW**.

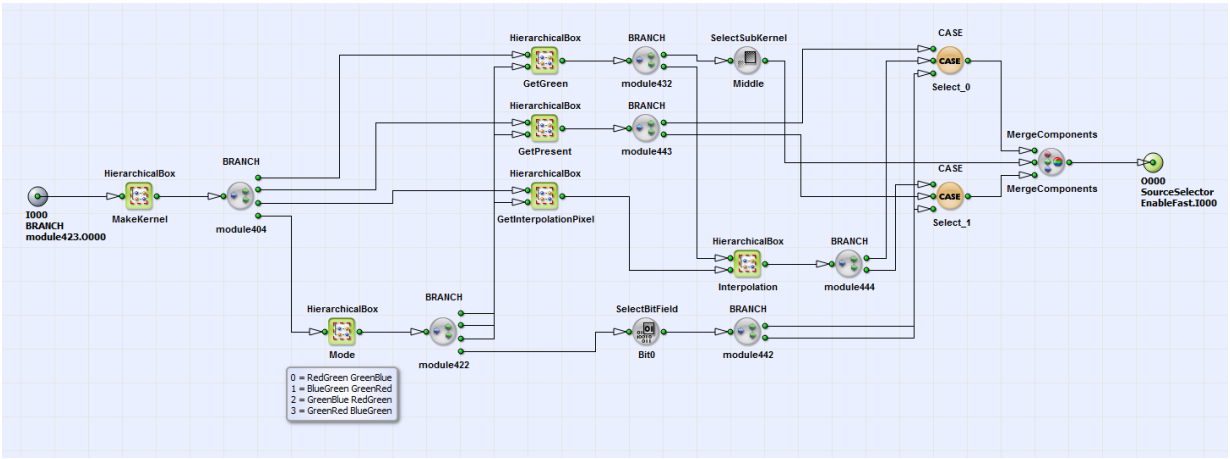



Figure 12.37. Content of HierarchicalBox **DeBayerEnhancedRAW**

In the HierarchicalBox **MakeKernel** a 6x1 Kernel is created. The kernel components are the two 8 bit color components of the current pixel value ("Enhanced Raw" mode!) with its next neighbors. See for visualization of the kernel components order Fig. 12.38 for an example Bayer pattern of "RedFollowedByGreen_GreenFollowedByBlue".

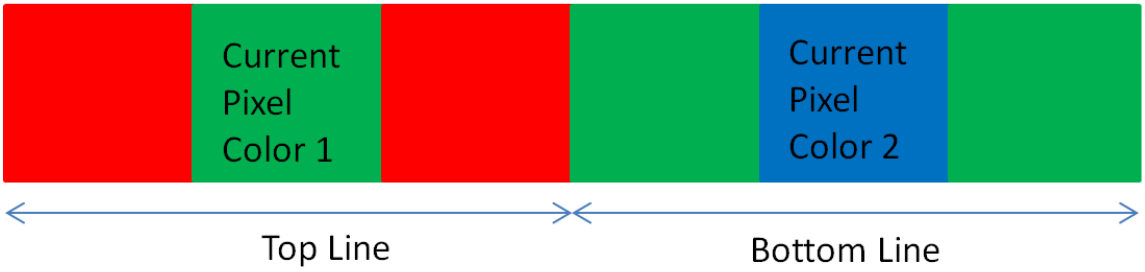



Figure 12.38. Kernel components created in HierarchicalBox **MakeKernel**

In the box **Mode** you can choose the Bayer pattern in setting the operator **Const_Mode** to a value between 0 and 4. See for more information on the corresponding Bayer pattern the comment box beside **Const_Mode**. As more comfortable alternative you can choose the Bayer pattern of your camera with "right-mouse-click" on the box **DeBayer** under "Properties" on the top level of the design as described above. In the boxes **GetGreen** and **GetPresent** the green component and the current color, which is red or blue, is selected for each pixel. In the box **Interpolation** the missing third color component red ($R_{i,j}$) or blue ($B_{i,j}$) at pixel position $i,\ j$ is interpolated according to [Bas13]:

$$R_{i,j} = \frac{R_{i,j-1} + R_{i,j+1}}{2} + \frac{-G_{i,j-1} + 2G_{i,j} - G_{i,j+1}.}{2} \tag{12.13}$$

The color components are finally merged together with the operator **MergeComponents**. If you choose the "Bayer Raw" mode of your camera you can choose for DeBayering of the color components box **DeBayerRAW** (see Fig. 12.36). See the content of box **DeBayerRAW** in Fig. 12.39.

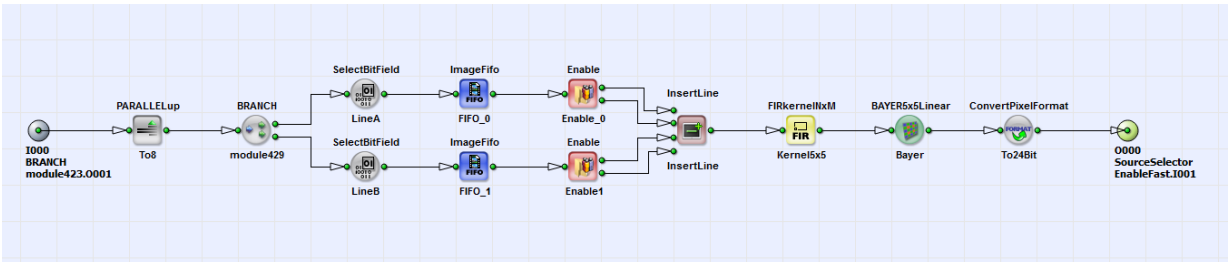
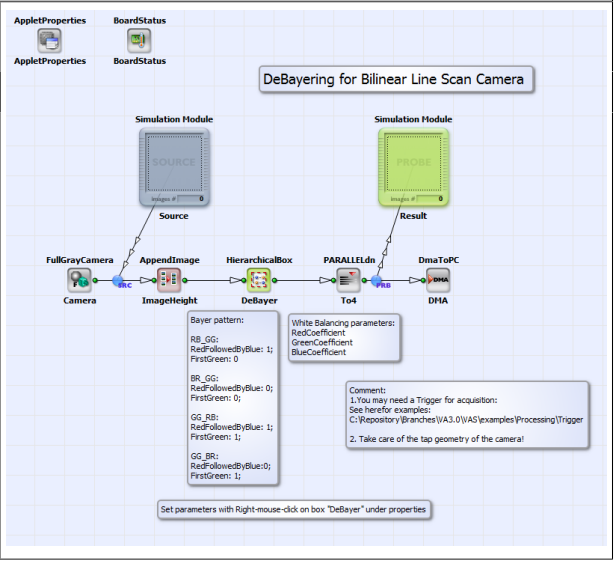



Figure 12.39. Content of HierarchicalBox **DeBayerRAW**

Here the DeBayering of the color components is performed with the operator **BAYER5x5Linear** on a 5x5 kernel around the current pixel. The color interpolated image is then transmitted via DMA to PC as shown in the basic design structure in Fig. 12.35.

### 12.4.1.10. Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern RedFollowedByBlue GreenFollowedByGreen

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \BilinearBayer_RB_GG.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>The example shows the de-mosaicing of a Bayer RAW pattern of a bilinear line scan camera with color pattern Red/BlueFollowedBlue/ Red_GreenFollowedGreen in "DeBayer Enhanced" and "DeBayer Fast" (is "Bayer Raw") mode. |  |

In the Visual Applets design example "BilinearBayer_RB_GG.va" the Bayer demosaicing for a bilinear line scan camera with colors red followed by blue or vice versa in the first sensor (top) line and colors green in the second (bottom) sensor line (or vice versa) is implemented. See Fig. 12.40 for the visualization of the sensor layout.

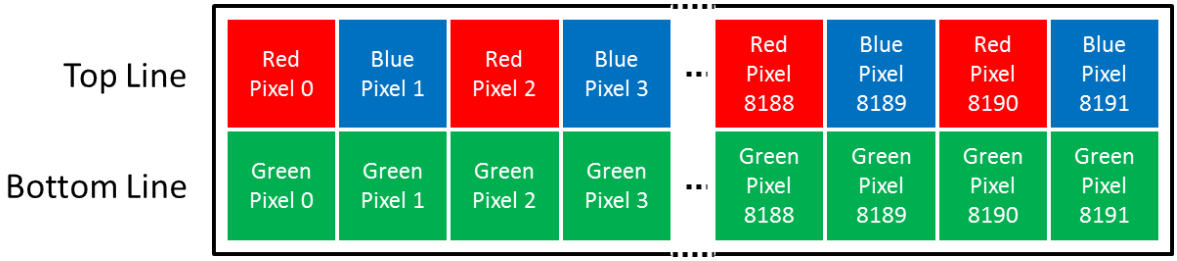



Figure 12.40. Sensor layout of a bilinear line scan camera with color pattern Red/BlueFollowedByBlue/ Red_GreenFollowedByGreen

The Bayer demosaicing for this sensor layout is implemented in the applet for two camera modes: the "DeBayer Fast" mode and the "DeBayer Enhanced" mode. In the first case the motion between two acquisitions is two sensor lines, in the second case one line. Due to this, one color component is acquired for each pixel in the "DeBayer Fast" mode and two color components (red and green or blue and green) per pixel in the "DeBayer Enhanced" mode. In Fig. 12.41 you can see the basic structure of the design "BilinearBayer_RB_GG.va".

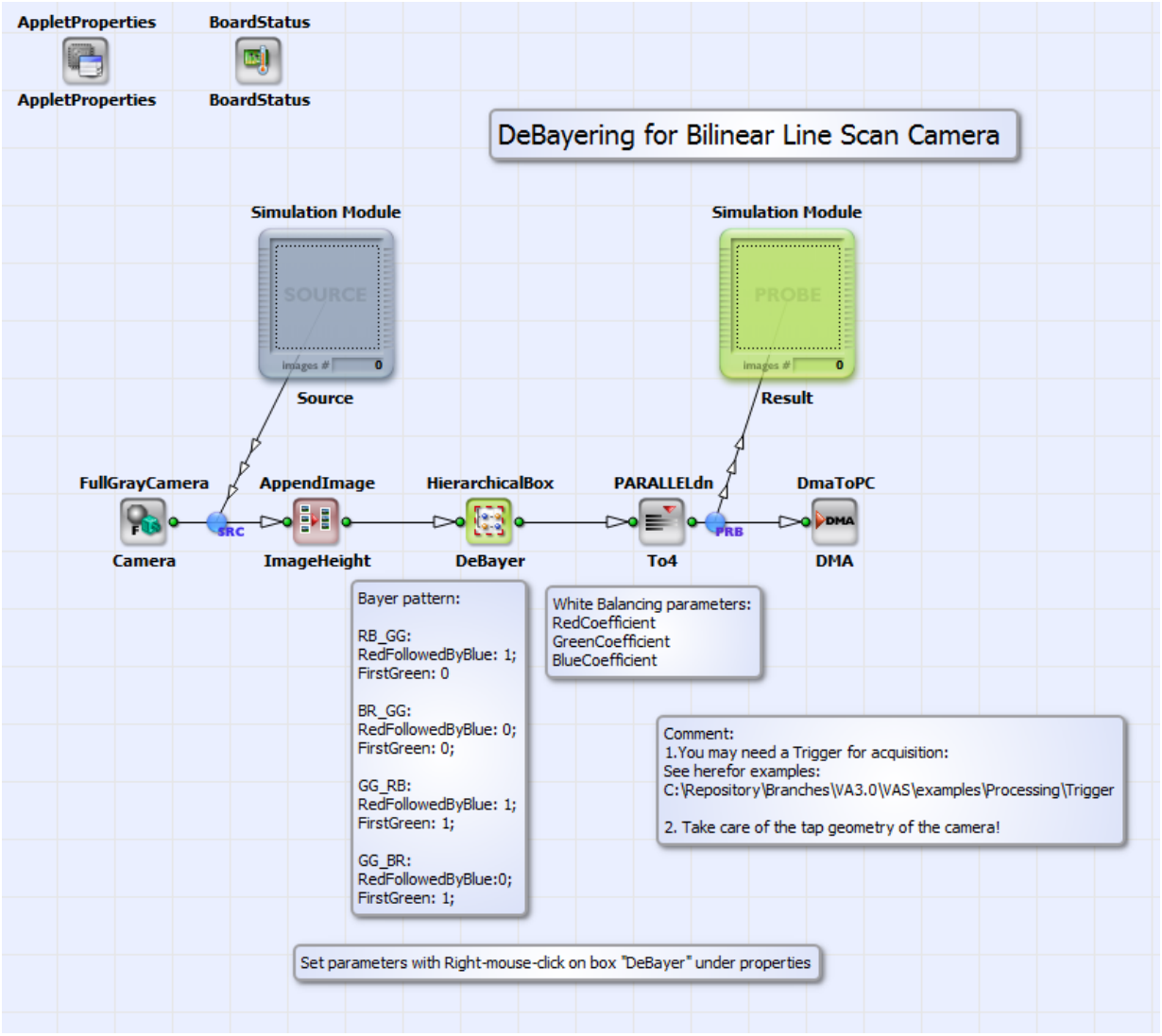



Figure 12.41. Basic design structure of "BilinearBayer_RB_GG.va"

An image acquired in "DeBayer Fast" or "DeBayer Enhanced" mode by a bilinear line scan camera in Camera Link Full configuration (see camera interface operator **FullGrayCamera**) is demosaiced in the HierarchicalBox **DeBayer**. With "right-mouse-click" on this box you can choose the Bayer pattern and the white balancing coefficients for red, green and blue under "Properties". The fully color interpolated rgb image is transmitted to PC via DMA. The default platform for this design is microEnable 5 marathon VCL, but it can easily be adapted to another platform. For the design you can find test images for the "DeBayer Enhanced" mode and the "DeBayer Fast" mode under \examples\Processing\Color\Bayer \TestImagesBilinearBayer. The content of box **DeBayer** is shown in Fig. 12.42.

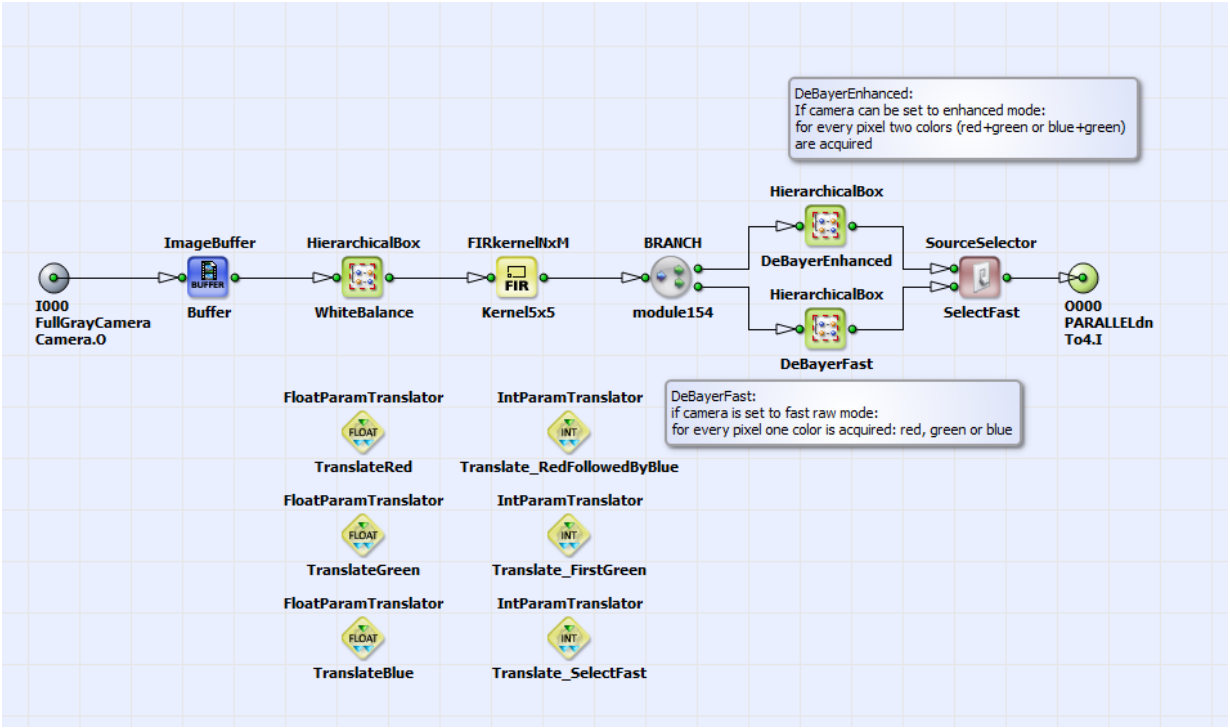



Figure 12.42. Content of HierarchicalBox **DeBayer**

In the HierarchicalBox **WhiteBalance** a white balancing of the red, green and blue components is performed. You can choose the white balancing coefficients directly with "double-mouse-click" on the operator **WhiteBalanceBayer** in this box or with "right-mouse-click" on the box "DeBayer" as described above. The operator **FIRkernelNxM_Kernel5x5** creates a 5x5 pixel kernel around the current pixel. With the operator **SourceSelector_SelectFast** you can choose the DeBayer mode **DeBayerEnhanced** or **DeBayerFast**. In Fig. 12.43 you can see the content of box **DeBayerEnhanced**.

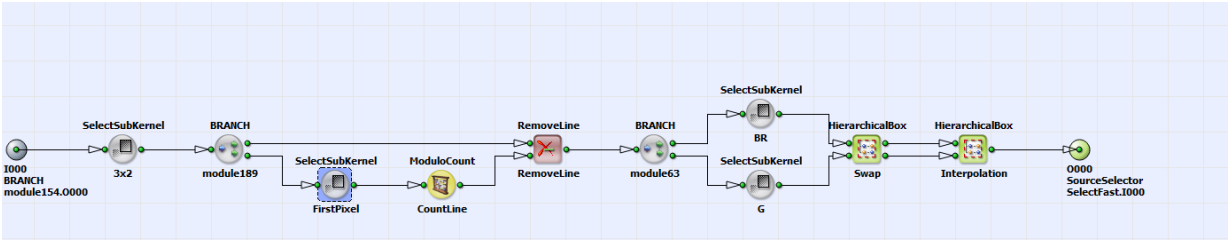



Figure 12.43. Content of HierarchicalBox **DeBayerEnhanced**

From the 5x5 kernel, a 3x2 sub kernel is selected. In the first row the red and blue values and in the second the green color values (or vice versa) are located. See for visualization of the kernel components order Fig. 12.44 for an example Bayer pattern of "RedFollowedByBlue_GreenFollowedByGreen".

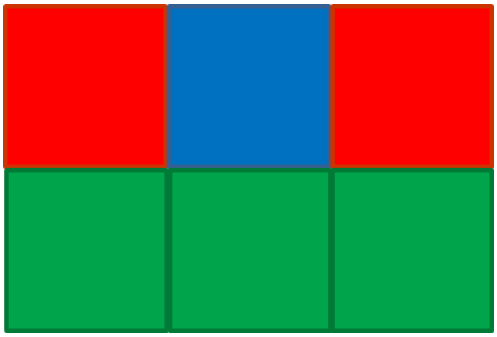


Figure 12.44. Kernel components selected in HierarchicalBox **DeBayerEnhanced** by operator **SelectSubKernel3x2**

With the operator combination **SelectSubKernel-FirstPixel**, **ModuloCountLine** and **RemoveLine** redundant image lines are removed. (The line redundancy occurs from the kernel creation.) With the operators **SelectSubKernel-BR** and **SelectSubKernel-G** the red and blue kernel components are separated from the green kernel components. If the Bayer pattern has the green values in the first kernel row, the components can be swapped in the HierarchicalBox **Swap** with the constant **IsBRthenGG**. The constant is set automatically, if you have chosen the Bayer pattern with "right-mouse-click" on the box "DeBayer" as described above. In the box **Interpolation** the missing third color component red ($R_{i,j}$) or blue ($B_{i,j}$) at pixel position $i,\ j$ is interpolated according to:

$$R_{i,j} = \frac{R_{i-1,j} + R_{i+1,j} - 4G_{i-1,j+1} + 8G_{i,j} - 4G_{i+1,j+1}}{2} \tag{12.14}$$

where $G_{i,j}$ are the green color components at pixel position $i,\ j$. The color components are finally merged together with the operator **MergeComponents**.

If you choose the "Bayer Raw" (also called "DeBayer Fast") mode of your camera, for each pixel only one color is acquired, due to the doubled line speed in comparison to the method described above. For this camera mode you can choose for DeBayering of the color components box **DeBayerFast** (see Fig. 12.42). You can see the content of box **DeBayerFast** in Fig. 12.45.

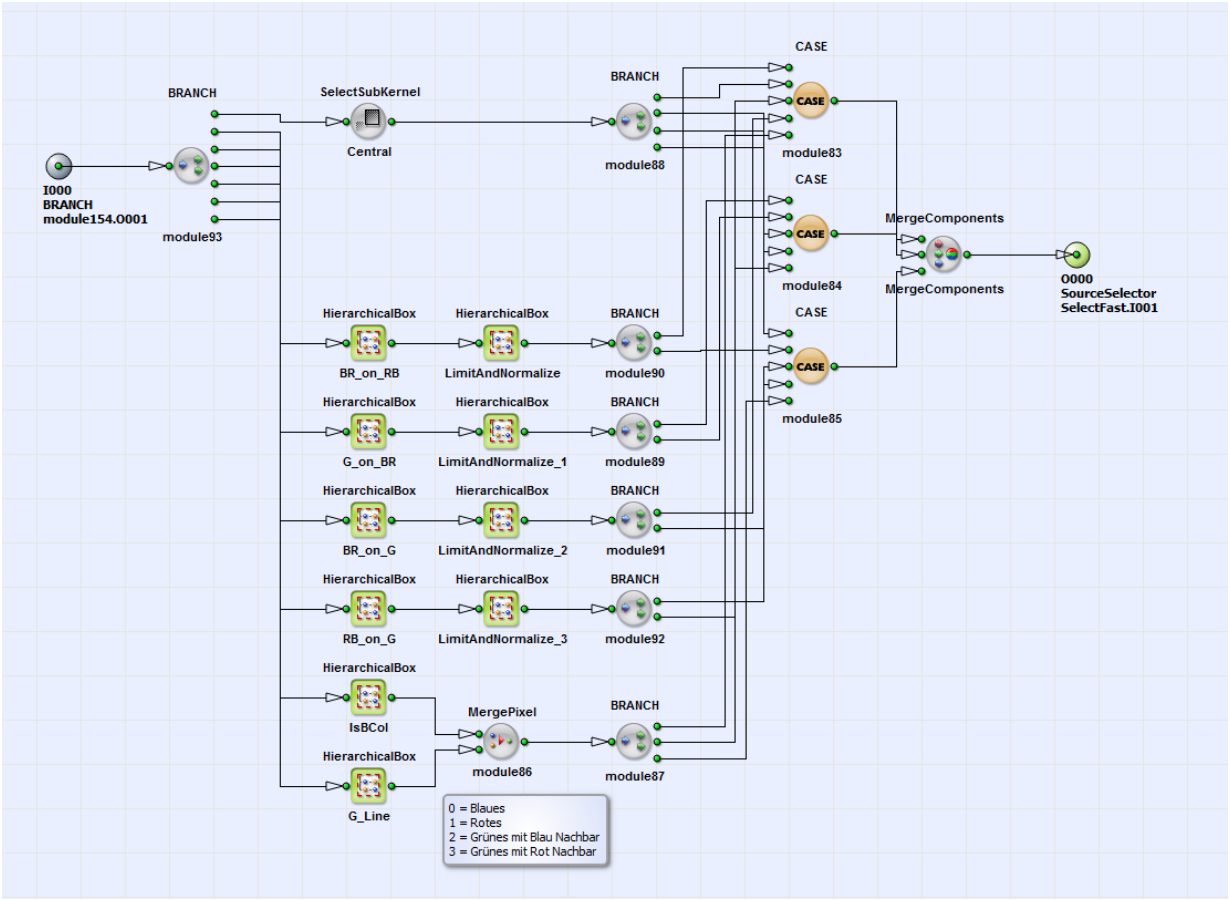



Figure 12.45. Content of HierarchicalBox **DeBayerFast**

Here the DeBayering of the color components red $R_{i,j}$, blue $B_{i,j}$ or green $G_{i,j}$ at pixel position $i$, $j$ is performed in the following way. The red or blue color components on a blue or red pixel are interpolated according to:

$$R_{i,j} = \frac{8B_{i,j} + 8R_{i-1,j} + 8R_{i+1,j} - 2R_{i,j-2} - 2R_{i-2,j} - 2R_{i,j+2} - 2R_{i+2,j}.}{16} \tag{12.15}$$

The green color components on a red or blue pixel are interpolated according to:

$$G_{i,j} = \frac{4R_{i,j} + 6G_{i,j-1} + 6G_{i,j+1} + G_{i-1,j-1} + Gi + 1, j - 1 + G_{i-1,j+1} + G_{i+1,j+1} - R_{i,j-2} - R_{i-2,j} - 2R_{i,j+2} - R_{i+2,j}}{16} \tag{12.16}$$

The red or blue color components on a green pixel in a red or blue pixel column can be calculated as:

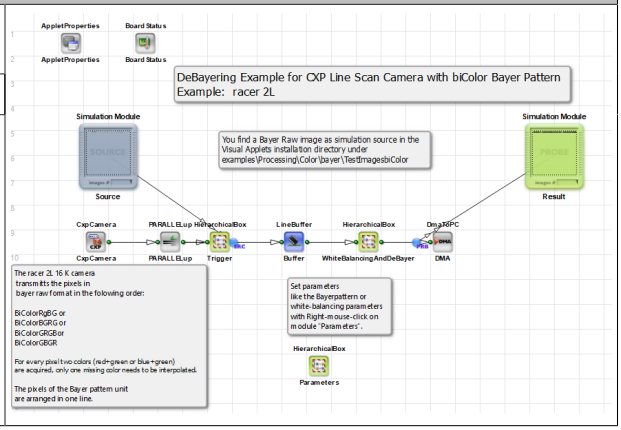
$$R_{i,j} = \frac{8R_{i,j-1} + 8R_{i,j+1} + 8G_{i,j} - 4G_{i-1,j} - 4G_{i+1,j}.}{16} \tag{12.17}$$

Finally the red or blue color components on green pixel in a blue or red pixel column are evaluated as:

$$R_{i,j} = \frac{4R_{i-1,j-1} + 4R_{i+1,j-1} + 4R_{i-1,j+1} + 4R_{i+1,j+1} - 4Gi - 1, j + 8G_{i,j} - 4G_{i+1,j}.}{16} \tag{12.18}$$

For each pixel the evaluated red, green and blue color components are merged together with the operator **MergeComponents** (see Fig. 12.45) The complete image is finally transmitted via DMA to PC as shown in the basic design structure in Fig. 12.41.

### 12.4.1.11. Bayer DeMosaicing a Line Scan Camera with Bi-color Bayer Pattern biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Color\Bayer \BiColorDeBayering_racer2L.vad<br>**Default Platform:** imaFlex CXP-12 Penta<br>**Short Description**<br>The example shows the de-mosaicing of a Bayer RAW pattern for a CXP-12 line scan camera with bi-color Bayer pattern. |  |

The example shows the de-mosaicing of a Bayer RAW pattern of a CXP-12 line scan camera with bi-color Bayer pattern: biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR, for example the racer 2 L camera. In addition, the example contains a line scan trigger module and a white balancing module.

## 12.4.2. Color Plane Separation

In machine vision a basic requirement is to separate color components in different images. This could be for example the separation of RGB into it's components, but as well the separation of HSL or XYZ etc. In the following, some examples for color separation are shown. All examples use a RGB area scan camera for input. Of course, the examples can be applied to line scan cameras as well. Moreover, an antecedent Bayer De-Mosaicing or a color space conversion is likely and can be easily added to the examples.

The examples assume that the color separation is required for post processing on the host PC. Therefore, the examples focus on color separated DMA transfers. If you need a processing on color separated data in the FPGA, the examples can also be applied to your requirements.

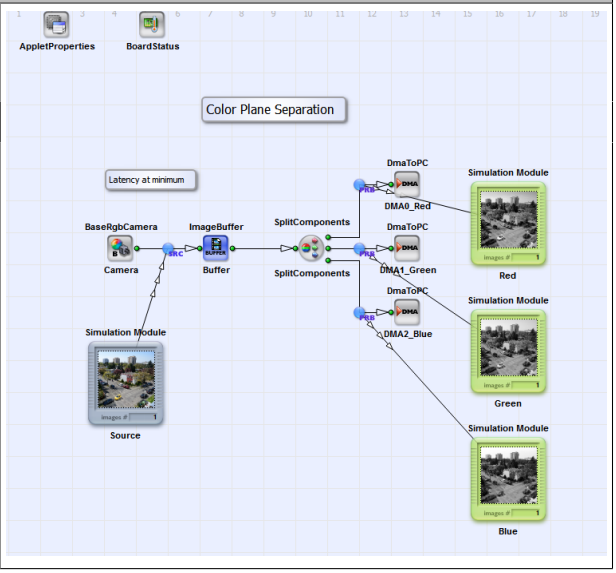
Several examples are shown. The result of all examples is the same. However, the implementations are completely different and all have their pros and cons. You will need to select the example which fits best to your requirements. The following table lists all examples and some of the properties.

| Example Name | Latency | DRAM | No. of Block RAM | DMA Output |
|---|---|---|---|---|
| Three DMAs | No additional latency | 1 DRAM buffer | low | 3 DMA. One for each color component. |
| Sequential with 3 buffers | Latency minimized to theoretical minimum. | 3 DRAM buffers | low | 1 DMA. Sequential color component output. |
| Sequential with MultiROI Buffer | 1 frame latency and bandwidth limitation. | 1 DRAM buffer | normal | 1 DMA. Sequential color component output. |
| Sequential with pre-sorted MultiROI Buffer | 1 frame latency | 1 DRAM buffer | high | 1 DMA. Sequential color component output. |
| Sequential with advanced buffer usage | 1 frame latency | 1 DRAM buffer | low | 1 DMA. Sequential color component output. |

Table 12.4. Overview of Color Separation Examples

A full description of the properties shown in the previous table as well as full descriptions of the different methods are outlined in the following sections.

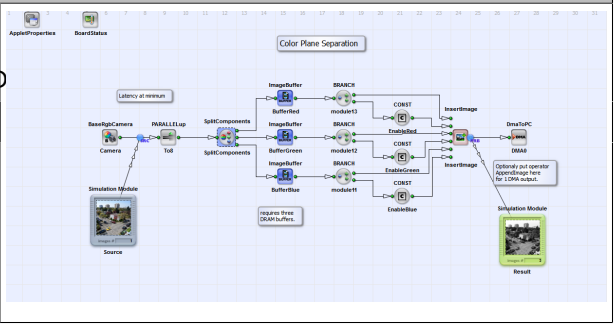
### 12.4.2.1. Color Plane Separation Option 1 - Three DMAs

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Color\ColorPlaneSeparation \ColorPlaneSeparation_Option1_ThreeDMA.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>RGB color plane separation examples. The components are split into three DMA output channels. | |

This example is very simple. The input RGB image is buffered and after, the color planes are split into three output links. These three links are transfered to the PC using three *DmaToPC* operators. An advantage of this solution is, that you have a minimized latency. The image is not buffered in the frame grabber. Operator ImageBuffer will only delay the image by one line if the host PC is fast enough. No additional delay is generated. Another advantage is that only one DRAM operator is required.

A disadvantage of the implementation is that three DMA output operators are required. These operators consume many FPGA logic resources of the frame grabber. If you do not require such a minimized latency, you should consider one of the solution presented in the following chapters.

### 12.4.2.2. Color Plane Separation Option 2 - Three Buffers, One DMA

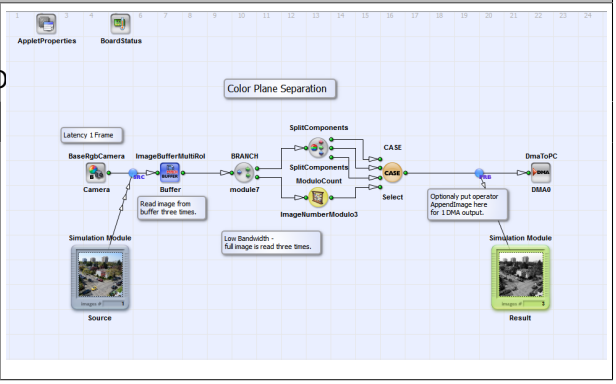
| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Color\ColorPlaneSeparation \ColorPlaneSeparation_Option2_ThreeBuffersOneD |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>RGB color plane separation example. The components are split and buffered in three frame grabber buffers. After that, the color planes are sequentially output using one DMA channel. | |

In this solution for RGB color plane separation, the color planes are output in sequential order over DMA channel. This is done by splitting the RGB input into its components and buffering the colors separately in three ImageBuffer modules. Using operator *InsertImage*, the color planes are read one after each other from the *ImageBuffer*s.

In this example, it is important to increase the parallelism before writing the data into the *ImageBuffer*. Otherwise you will get a bottleneck after the InsertImage module.

The advantage of this approach is that you will only need one DMA channel which reduces the required resources. Also, the latency is at its theoretic minimum for sequential output. The DMA transfer for the red channel immediately starts without delay. A drawback of this solution is that three frame grabber memory operators are required. As they are limited this solution is only useful if not more of these buffers are required.

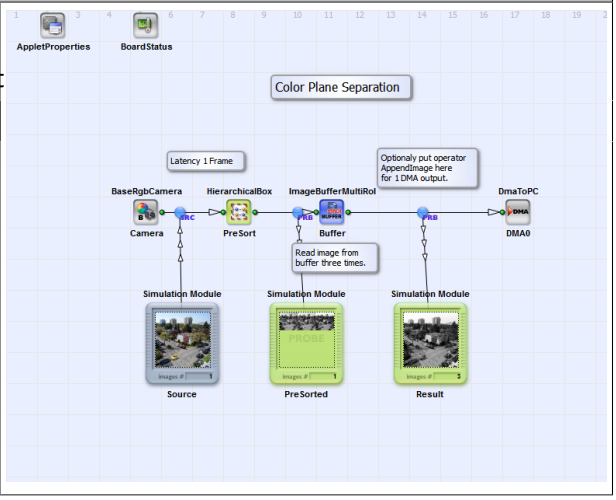
### 12.4.2.3. Color Plane Separation Option 3 - Sequential with Operator ImageBufferMultiRoI

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Color\ColorPlaneSeparation \ColorPlaneSeparation_Option3_SequentialMultiRO |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>RGB color plane separation example. A sequential DMA output of the three RGB color planes is performed. The sequence is generated by *ImageBufferMultiRoI*. This solution is not good as it limits the bandwidth. See the next example, for an optimized solution. | |

This solution is an intermediate step between the previous and the next example. Because of bandwidth limitations, it is not advised to implement this solution. However, it explains in a simple way on how to get the same output as the previous example with requiring only one frame grabber RAM. To sequentialize the image, we use operator *ImageBufferMultiRoI*. In this operator, we simply read the same image three times. After, we select the red component for the first image, the green for the second image and the blue for the third image. This can be easily done with the *ModuloCount* and the *CASE* operators. The *ModuloCount* is parameterized to count frame numbers modulo 3 i.e. {0, 1, 2, 0, 1, 2, ...}.

As mentioned, this solution is not the optimum as it reduces the bandwidth. The reason is that we read the same input image three time, but using only one of the three components. Thus in every read cycle, we discard two third of the the data. The next example will show a similar solution which overcomes this limitation.

### 12.4.2.4. Color Plane Separation Option 4 - Sequential with Operator ImageBufferMultiRoI and a pre-sort of the Color Planes

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Color\ColorPlaneSeparation \ColorPlaneSeparation_Option4_SequentialPreSort |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>RGB color plane separation example. A sequential DMA output of the three RGB color planes is performed. The sequence is generated by *ImageBufferMultiRoI*. An additional pre-sorting optimizes the bandwidth and solves the problem presented in the previous example. | |

This example is similar to the previous example. In contrast, we do not have a bandwidth limitation in this case. The idea is the same. By use of operator *ImageBufferMultiRoI*, we sequentialize the three color components. In contrast to the previous example, we do not separate the colors after the buffer. Instead we separate the colors before writing them to the buffer. This is done in the the *HierarchicalBox* PreSort which content is shown in the following figure.

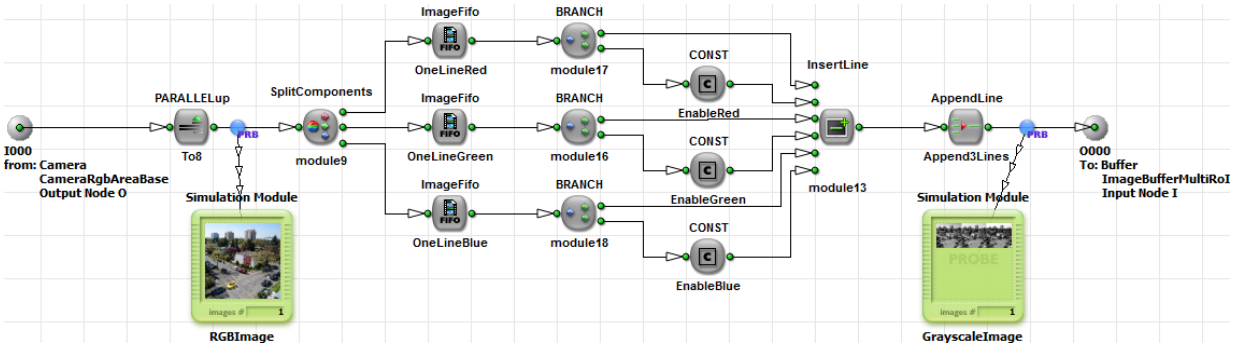



Figure 12.46. Pre-Sorting of Color Components

The idea of this approach is to separate the color components before writing them into the buffer. This is done for every image line. Thus we split the color components into three links. These resulting grayscale links now contain the image lines of the input images. Next, the lines are multiplexed and appended. The result is that we have thre grayscale images in one large image. The red component is on the left, the green in the middle and the blue on the right. Have a look at the simulation results in the next figure to understand the implementation.

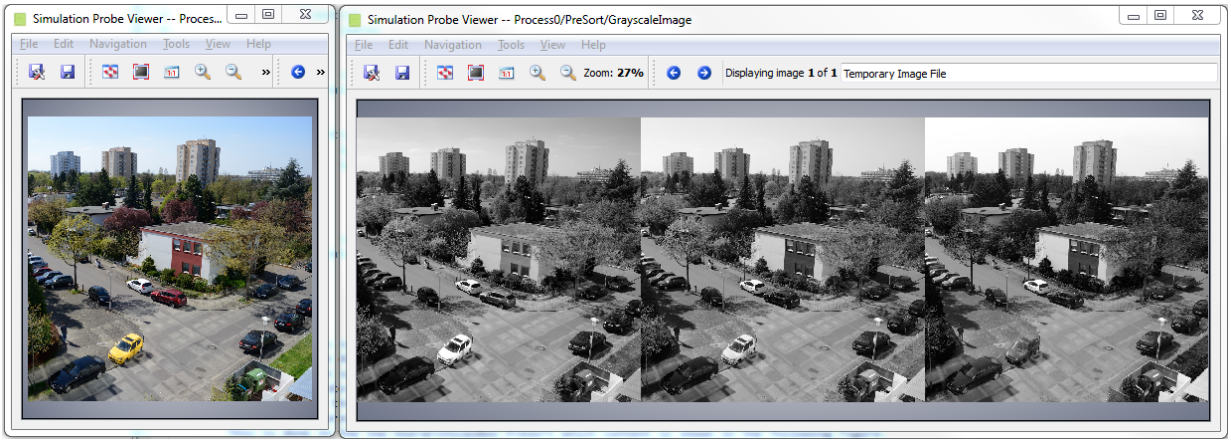



Figure 12.47. Simulation Result of Pre-Sorted Color Components

In the *ImageBufferMultiRoI*, we now have to read the left part first. After that the middle part and finally the right part. Thus, we need to set a different *XOffset* for each ROI as shown in the following.

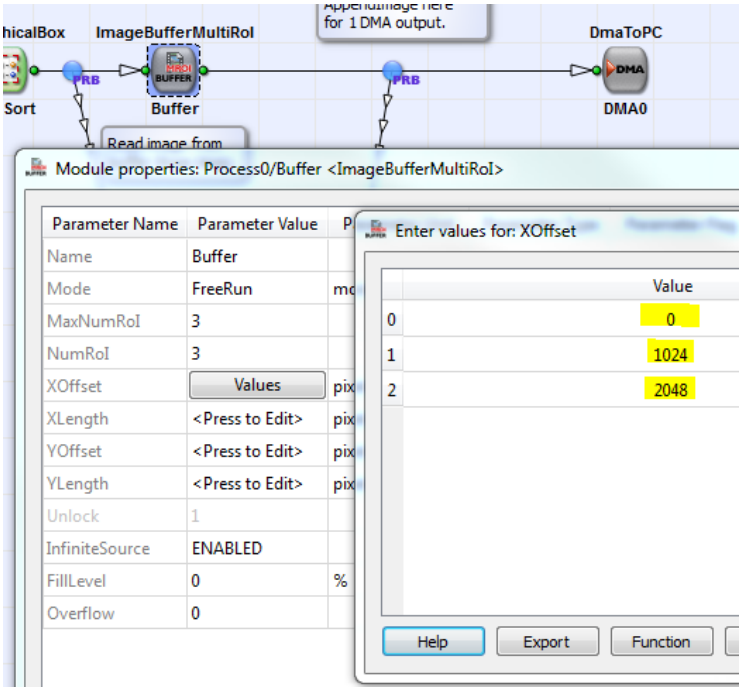
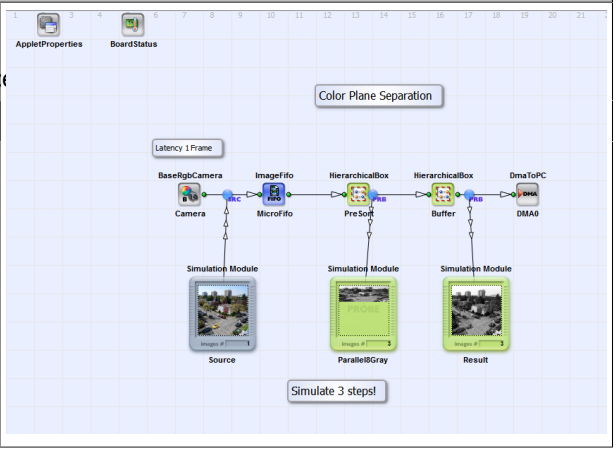



Figure 12.48. Parameter Setting for XOffset of the ImageBufferMultiRoI Operator

When using, do not forget to increase the parallelism before separating the components as shown in the example. This is required to avoid a bottleneck after the InsertLine module. We need at least a three times higher parallelism than the input as we transfer grayscale 8 bit image and not 24 bit RGB images anymore.

This solution does require only one DRAM operator and does not limit the bandwidth. This is an optimized solution. However, some block RAM is required to buffer the separated component lines in the FIFOs. For small images, this should not cause a problem. However, if you have large image widths such as 16384 pixel per line, many block RAM is required. This last drawback is solved with the next example.

### 12.4.2.5. Color Plane Separation Option 5 - Sequential Output with Advances Processing

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Color\ColorPlaneSeparation \ColorPlaneSeparation_Option5_SequentialAdvanc |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with *FrameBufferRandomRead*. | |



This is the most advanced example for color plane separation in VisualApplets. It is focused on a minimum resource usage and maximum bandwidth. The previous example used a pre-sorting to separate the color components so that we could individually read them from the buffer. However,

some block RAM is required for the FIFOs in the pre-sorting. In this approach, we use a pre-sorting, too. In contrast, we collect 8 successive pixels of each color component only before switching to the next component. In the previous example, we collected a full line of one component before switching to the next line. This modification results in that we do not need FIFOs anymore. However, the *ImageBufferMultiRoI* cannot be used in this case. Instead, we have to use a *FrameBufferRandomRead*.

The pre-sorting is simple. First, we increase the parallelism to eight. Next, the components are split. This results in three links with parallel eight each. MergeParallel will now put the eight pixel of the components in a sequence. We will therefore get a sequence: R0 to R7, G0 to G7, B0 to B7, R8 to R15, G8 to G15, B8 to B15, ...

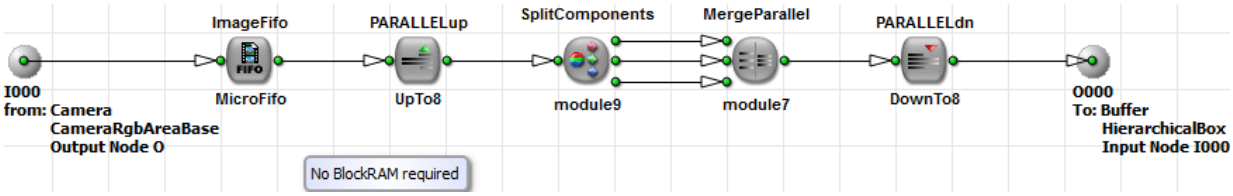



Figure 12.49. Pre-Sorting for Color Separation by collecting eight successive pixel of the same component.

The FIFO at the input is required because of *PARALLELdn* from 16 to 8. This FIFO is required only to avoid a DRC level 2 error. In fact, it will not need to buffer data, so that we set it to a very small size of 2 pixel only.

After we have pre-sorted the pixel, they are buffered in a *FrameBufferRandomRead* operator. This operator has read row and column address inputs and allows to randomly read the data.

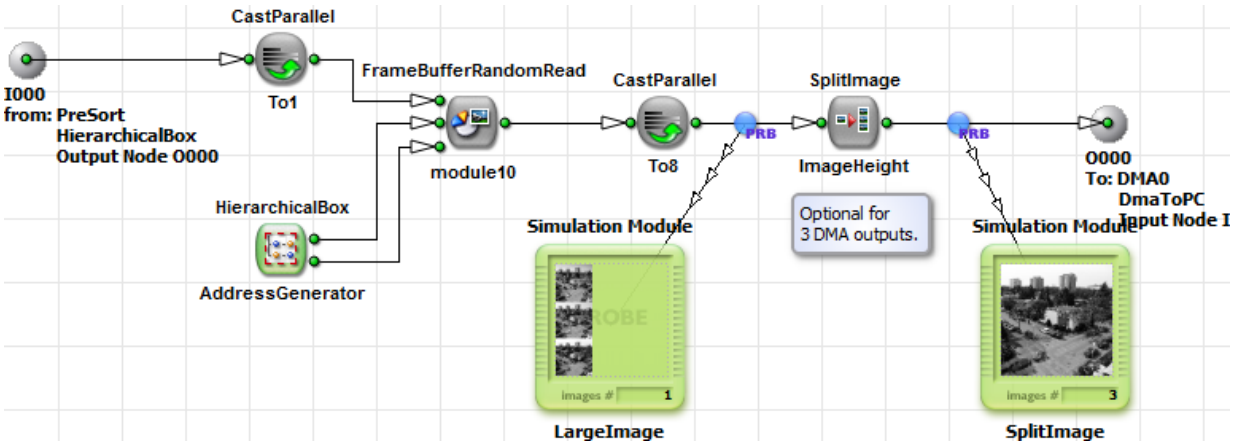



Figure 12.50. Color Separation with FrameBufferRandomRead

The operator can only be used with a parallelism of one. Therefore, we use *CastParallel* to cast from parallelism 8 at 8 bit per pixel to parallelism 1 at 64 bit per pixel. That's why we collected eight successive pixel in the previous step.

To separate the color components we have to increase the addresses by two to jump over the unwanted colors. For the red component, the read addresses will therefore be: RColA(0) = 0, RColA(1) = 2, RColA(2) = 5, ... For the green component, the read addresses will be: RColA(0) = 1, RColA(1) = 3, RColA(2) = 6, ... Hierarchical box AdressGenerator does generate these read addresses. A CreateBlankImage operator defines the required image dimension. As we previously used *CastParallel* from 8 to 1, we have to use an image width divided by 8.

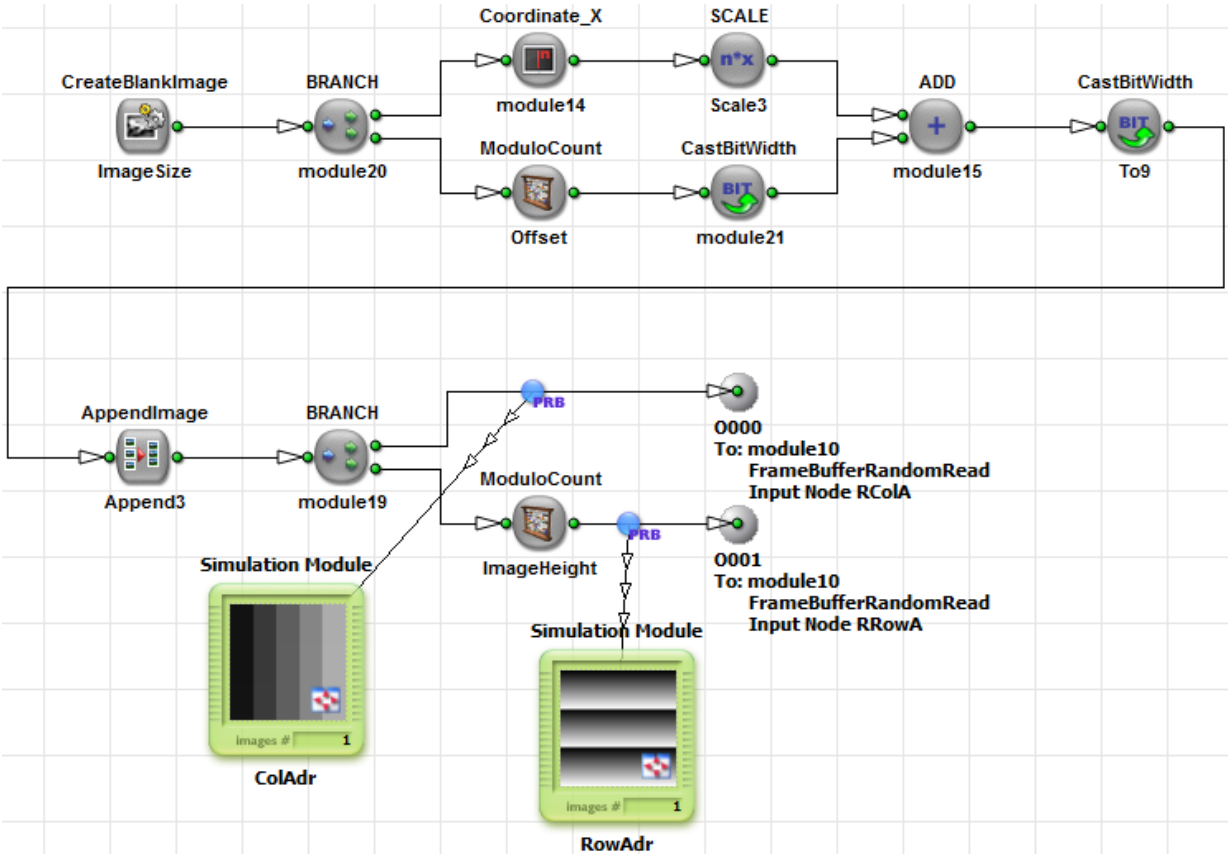



Figure 12.51. Address Generator for FrameBufferRandomRead Input

The column address is simply generated with scaling the *Coordinate_X* output by 3. Depending on the current component, we add 0, 1 or 2 as an offset to the values.

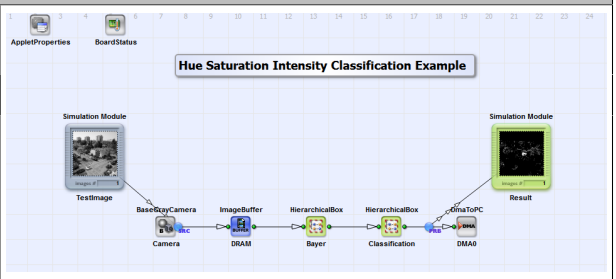
By use of *AppendImage*, the three address images are merged into one larger image. The row address is simply generated with a counter between 0 and the image height. In this case, we use a *ModuloCount* operator for this task. The addresses can now be fed into the *FrameBufferRandomRead* operator. Do not forget to use a parallelism two at the *CreatBlankImage* output to maximize the performance of FrameBufferRandomRead.

After the buffer, we obtain a large image where the first lines include the red image, after the image contains the green image and finally the blue. To split this large image into three separated DMA outputs, we use *SplitImage* and set the image height.

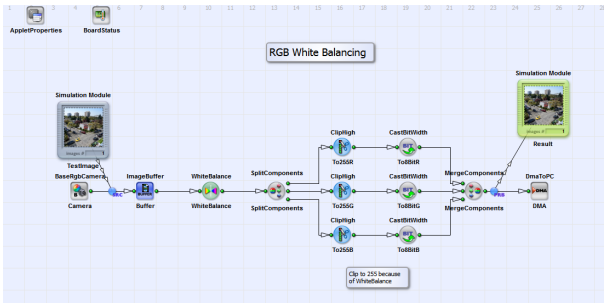
The easiest way to understand the implementation is by looking at the intermediate results step by step. Note that you will need to simulate three steps for one result image. That's because 3 images have to be generated by operator *CreateBlankImage*.

This solution is resource optimized, as only few block RAM resources are required. Moreover, the FPGA logic resources are efficiently used. The address generation only requires few resources.

### 12.4.3. HSL Color Classification

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Color \HSI_Classification\HSI_Classification.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Color Classification is very simple on HSL images. The applet converts the RGB image into an HSL image and performs a color classification. The hue is filtered using a lookup table. Moreover, the saturation and lightness is thresholded using custom threshold values. |  |

### 12.4.4. RGB White Balancing

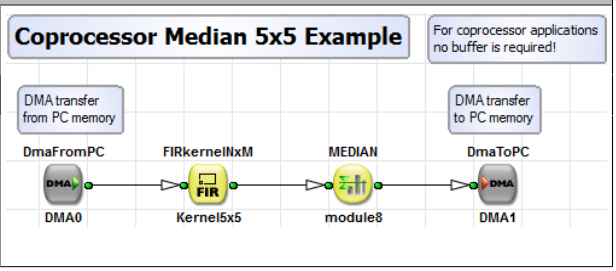
| Brief Description | |
|---|---|
| **File:** \examples\Processing\Color\Whitebalancing \RGBWhiteBalancing.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>The applet shows an example for white balancing on RGB images. |  |

For white balancing on Bayer RAW images see Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing'.

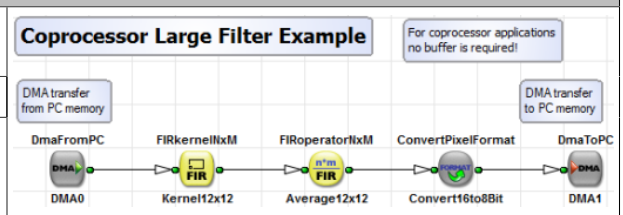
## 12.5. Co-Processor

In this section we provide two co-processor examples. For these no buffer is required. One design is a large 12x12 filter implementation, the second a 5x5 median filter algorithm.

### 12.5.1. Co-Processor Median Filter

| Brief Description | |
|---|---|
| **File:** \examples\Processing\CoProcessor \CoprocessorMedian\CoprocessorMedian.va<br>**Default Platform:** mE4VD1-CL only<br>**Short Description**<br>The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a median filter is calculated. |  |

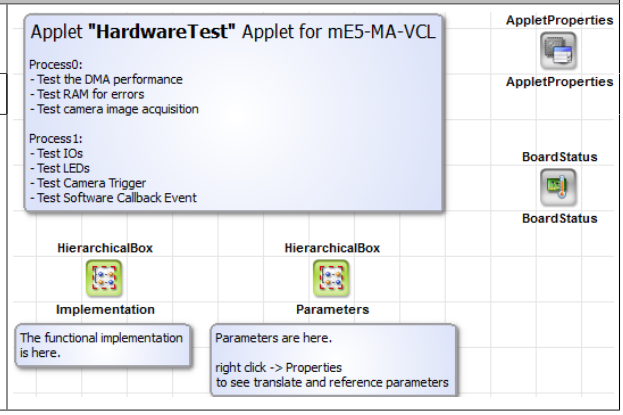
### 12.5.2. Co-Processor Large Filter Calculation

| Brief Description | |
|---|---|
| **File:** \examples\Processing\CoProcessor \LargeFilter\LargeFilter.va |  |
| **Default Platform:** mE4VD1-CL only | |
| **Short Description**<br><br>The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a large filter kernel is calculated. | |

# 12.6. Debugging and Test

Examples to assist in debugging of applets and test of hardware.

## 12.6.1. Hardware Test

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DebuggingAndTest \HardwareTest\HardwareTest.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs. This example extensively uses operators of the parameters library. | |

The example Hardware Test performs a hardware self test.

The applet comprises the following functions:

- DMA Performance test: Different image dimensions for varying memory sizes and interrupt rates
- RAM Test: Check for errors and processing
- Camera: Check camera port image acquisition
- Camera Trigger: Send trigger signals to camera
- GPIO: Monitor the GPIs and set the GPOs
- Event test: Generate a sofware callback event
- Monitoring: FPGA Temperature, Power, PoCL, ... (See *BoardStatus*)

The following diagram shows the functional blocks of the applet.

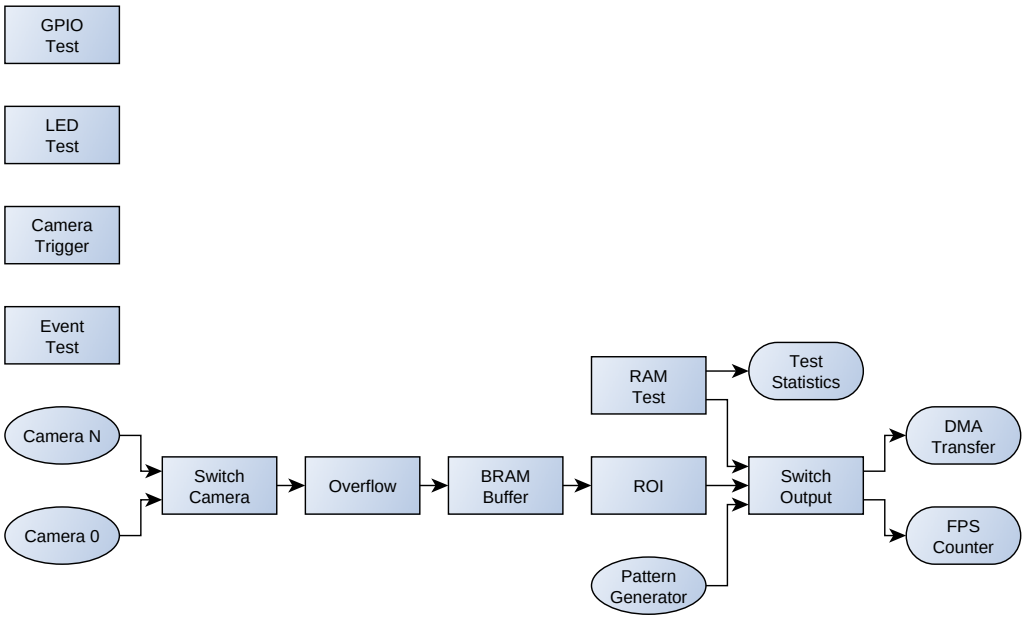



Figure 12.52. Block Diagram of the applet Hardware Test

The VisualApplets implementation is split into two processes. Process0 includes all image processing parts namely the DMA test, RAM test and camera acquisition. Process1 instead includes the parts which are independent of a running acquisition. As processes without DMA operators are immediately started when the applet is loaded, the functions here can be used independently of an acquisition. See Section 4.4.2, 'Processes without DMAs / Trigger Processes' for more information.

The following two screenshots show Process0 and Process1.

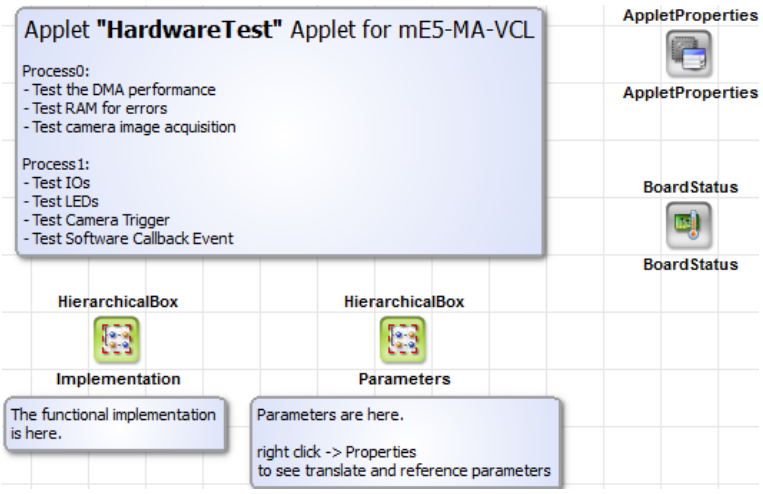



Figure 12.53. Hardware Test Process0

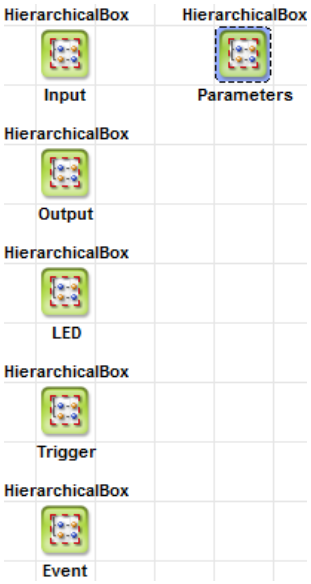



Figure 12.54. Hardware Test Process1

The applet is implemented by intensively using the operators of the *parameters* library for easy usage of the applet. The H-Box Implementation includes the functional parts. Parameters are listed in the properties of the H-Box Parameters for a separated view. This becomes useful when the applet is used in e.g. microDisplay as only easy to use parameters are listed as can be seen in the next screenshot.

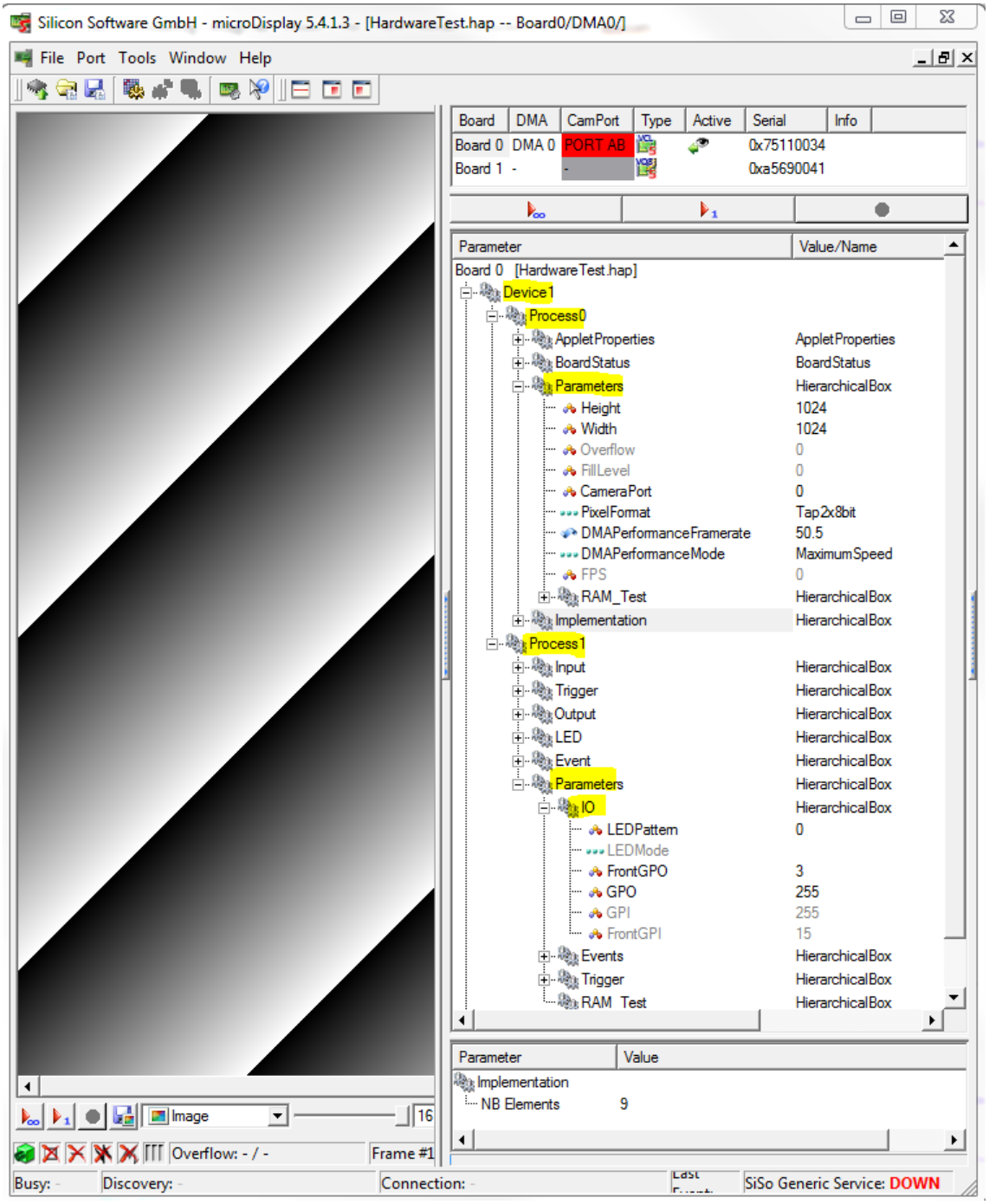



Figure 12.55. Applet Hardware Test use of Parameter Translates and References in microDisplay

The implementation of the applet is straight forward and can be easly understood by studying the VA file. However, we will have a closer look at some of the parmeter translations and references to understand their usage.

### 12.6.1.1. Implementation of Main Image Processing Part

The main image processing part contains the RAM test, DMA perfomance test and camera acquisition. The applet only allows to output one of the modes on a DMA channel at the same time. Thus, several *SourceSelector* operators are used. To avoid a parameterization specifying the correct input index we use a *EnumParamTranslator* operator for easy usage.

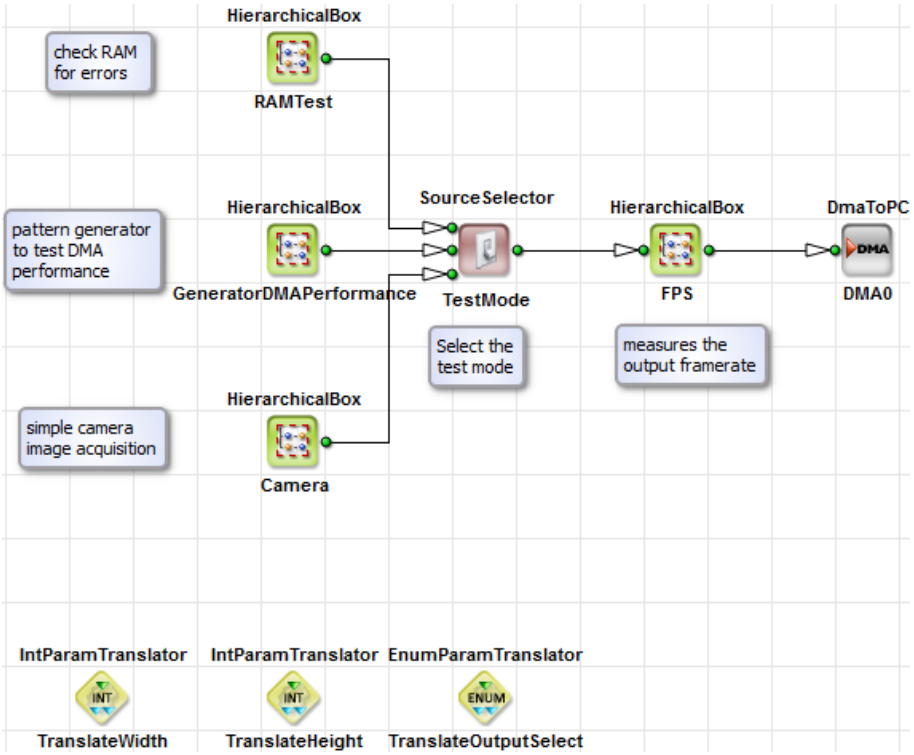
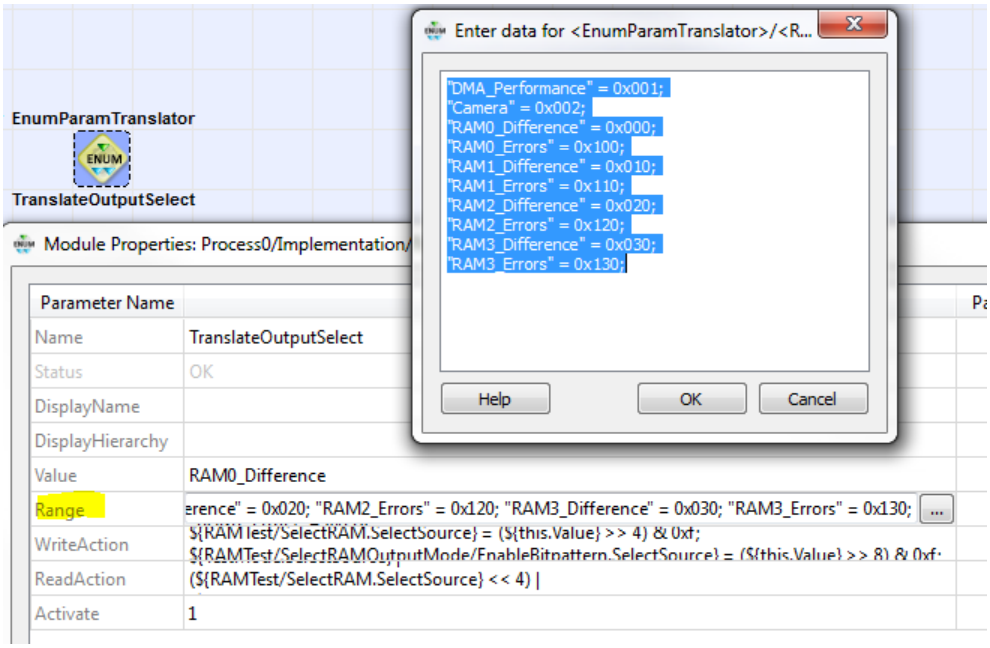



Figure 12.56. Applet Hardware Test Implementation for RAM Test, DMA Perfomance Test and Camera Acquisition

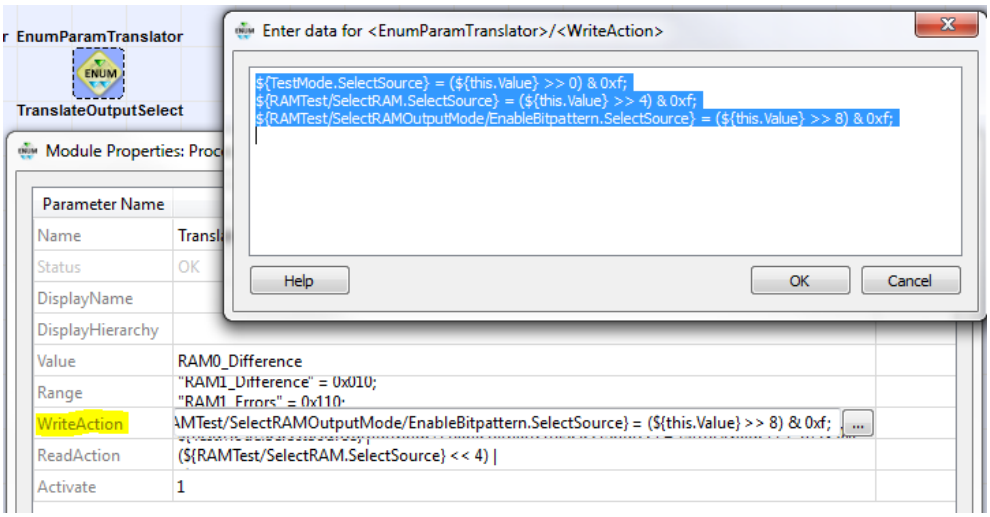
In the following we describe the parameter values of **`TranslateOutputSelect`** in detail.

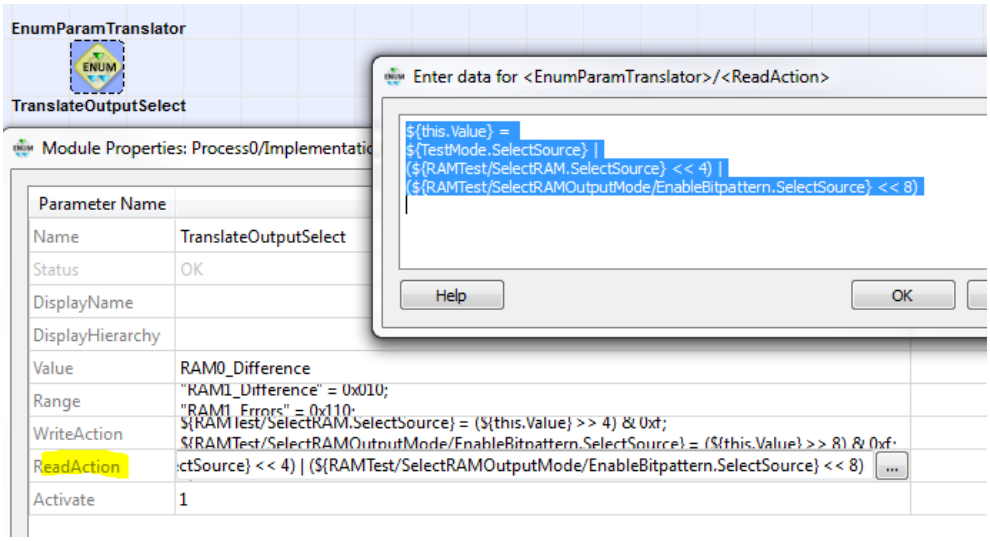
First, the enumeration values need to be defined.

```
"DMA_Performance" = 0x001;
"Camera" = 0x002;
"RAM0_Difference" = 0x000;
"RAM0_Errors" = 0x100;
"RAM1_Difference" = 0x010;
"RAM1_Errors" = 0x110;
"RAM2_Difference" = 0x020;
"RAM2_Errors" = 0x120;
"RAM3_Difference" = 0x030;
"RAM3_Errors" = 0x130;
```

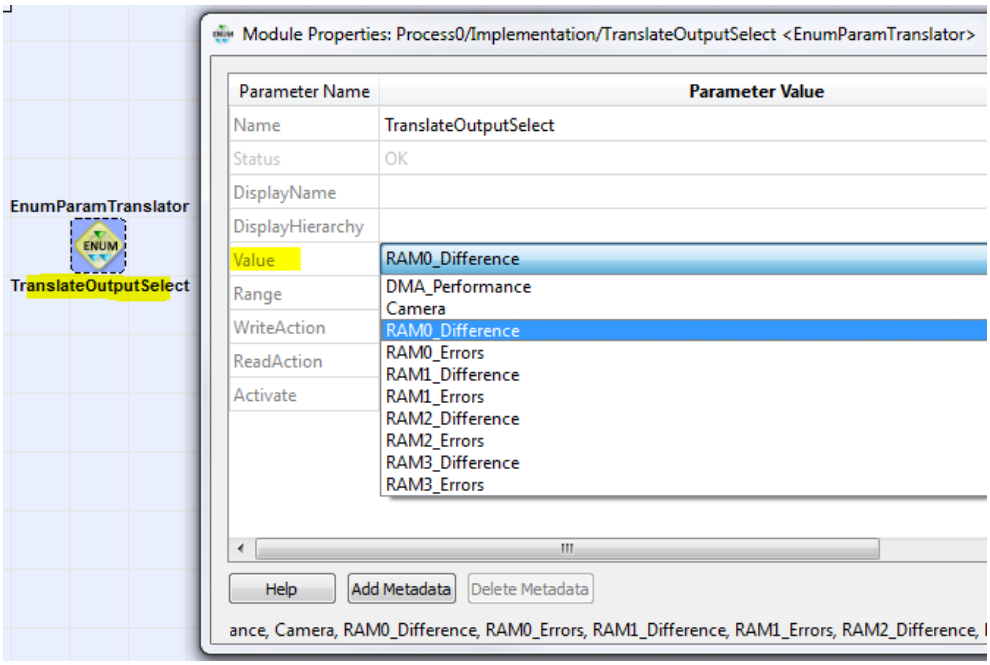



As you can see the values of the enumerations are defined as hexadecimal values. This allows an easy distribution to three *SourceSelector* modules. The lower four bit are used for module *TestMode.SelectSource*, bits 4 to 7 for *RAMTest/SelectRAM.SelectSource* and bits 8 to 11 for *RAMTest/SelectRAMOutputMode/EnableBitpattern.SelectSource*. This facilitates a very simple implementation for the parameters *WriteAction* and *ReadAction*.



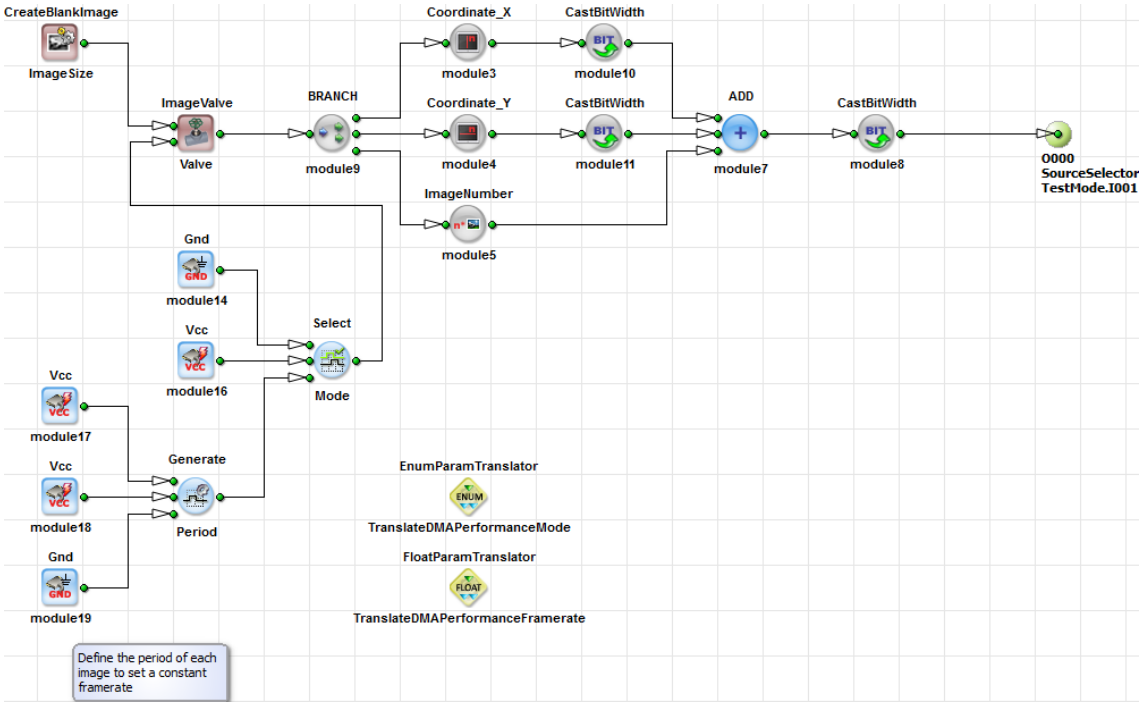



The translate value then can be used with the enumeration values. Changing the value will automatically change the parameter values of the *SourceSelector* modules.




## 12.6.1.2. Implementation of DMA Performance Pattern Generator

H-Box GeneratorDMAPerformance includes the pattern generator as well as an image valve to controll the output framerate. The pattern generation is very simple using a blank image generator as well as pixel and image counter to generate a rolling diagonal pattern.




The valve is controlled by a period generator operator *Generate*. The unit for the generator is given in ticks. As we want to specify a framerate with a time-base, we use translate operator *FloatParamTranslator*. Thus we use the operator to convert from frames per second to ticks. The following screenshot shows the implementation.

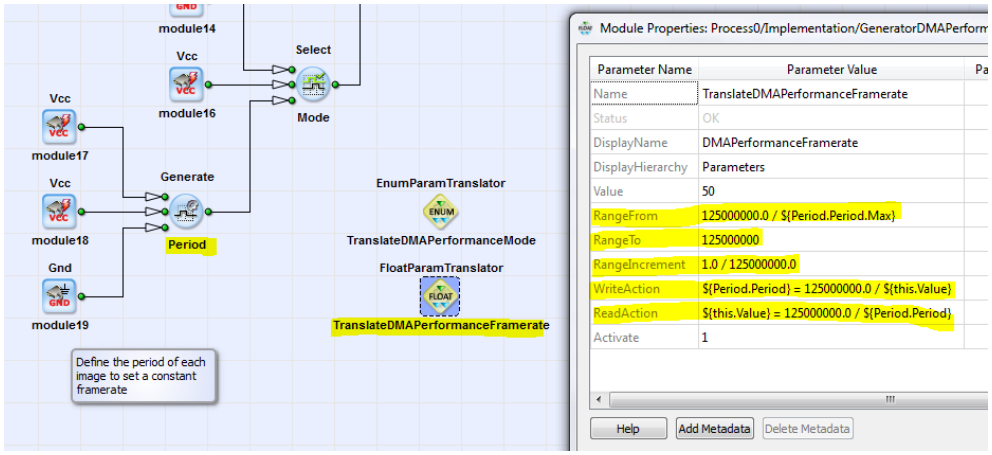



Figure 12.57. Use of *FloatParamTranslator* to Convert from Frames per Second to Ticks

## 12.6.1.3. Setting the Width and Height in the Example HardwareTest using operator *IntParamTranslator*

The same procdeure as shown in the previous sections is done for the image width and height. In this case we can use an operator *IntParamTranslator*. Here we can directly forward the values to the two modules in the design. Mind the given values for parameters *DisplayName* and *DisplayHierarchy*.

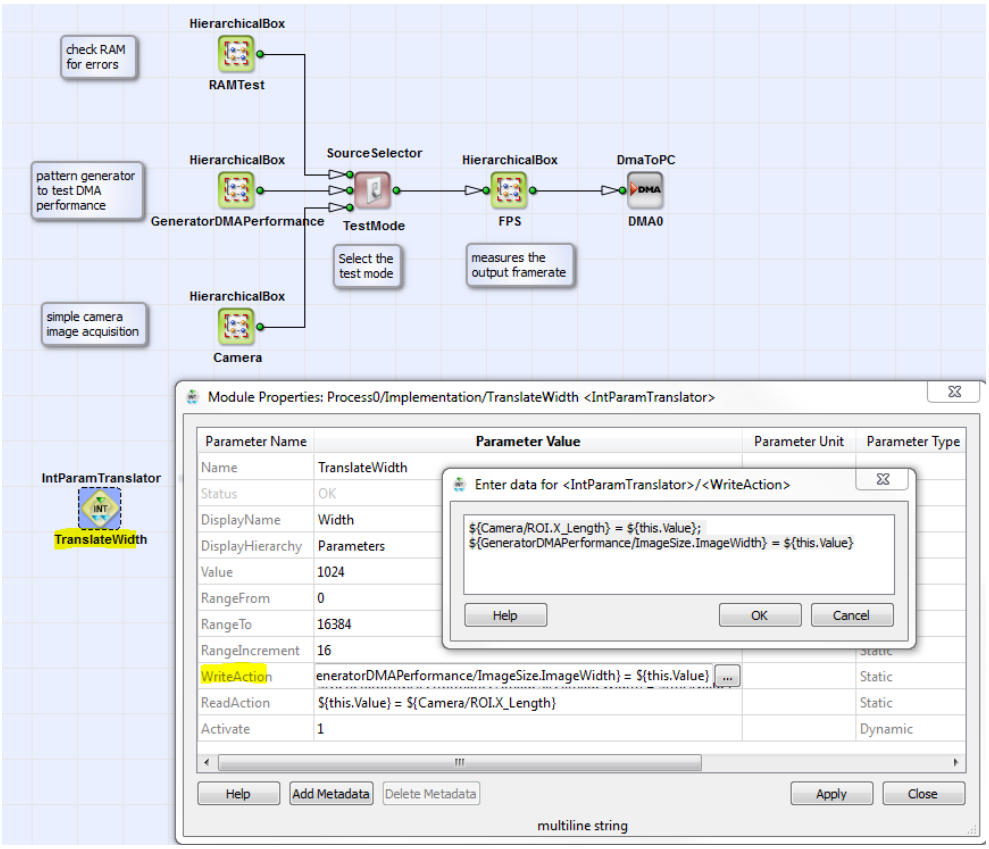
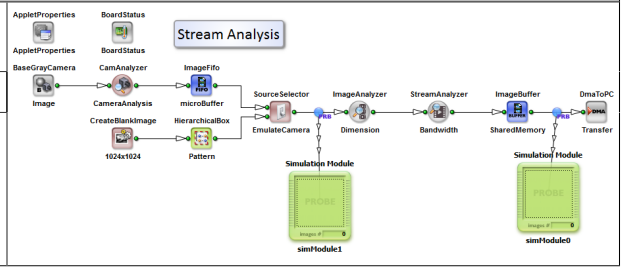


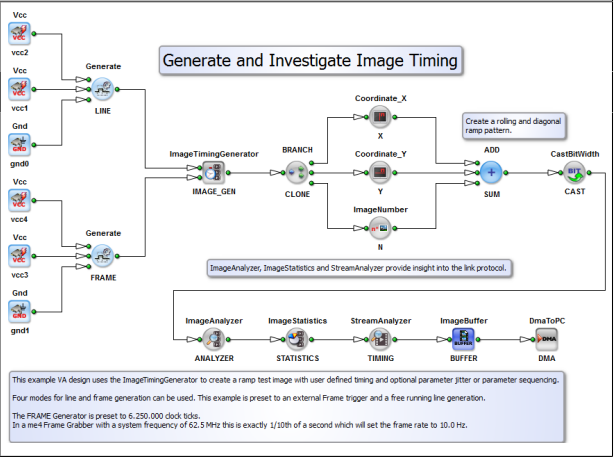

Figure 12.58. Use of *IntParamTranslator* for easy Setting of the Width and Height in the Applet

## 12.6.2. Image Dimension Test

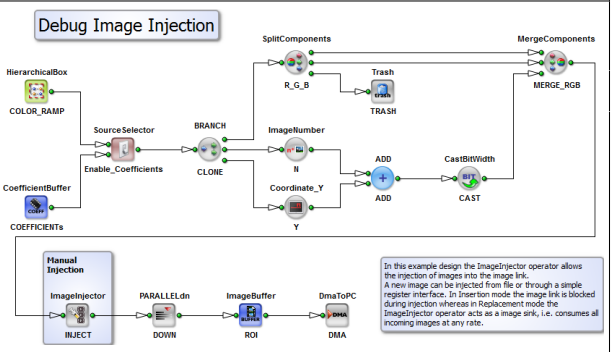
| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\DebuggingAndTest\ImageDimensionTest.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>The image dimension is measured and can be used to analyze the design flow. Especially the influence of the shared memory can be seen. | |



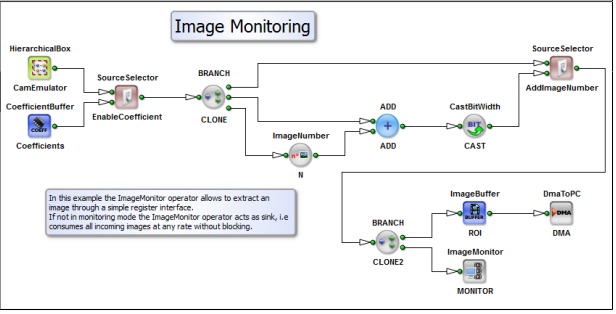
## 12.6.3. Image Timing Generator

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DebuggingAndTest\ImageTimingGenerator.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>This example design uses the ImageTimingGenerator to create a ramp test image with user defined timing and optional parameter jitter or parameter sequencing. Four modes for line and frame generation can be used. This example is preset to an external Frame trigger and a free running line generation. |  |

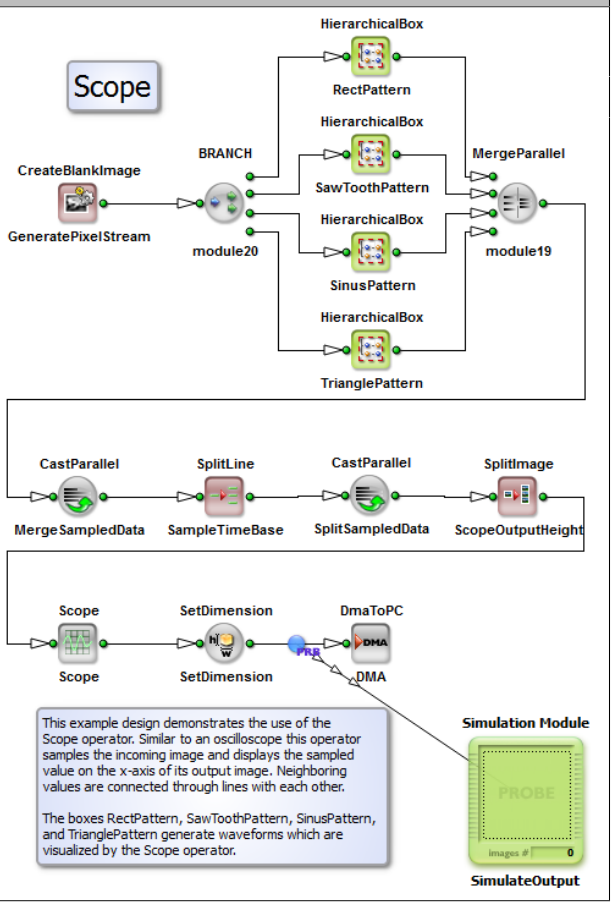
## 12.6.4. Manual Image Injection

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DebuggingAndTest\ImageInjector.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>In this example design the ImageInjector operator allows the injection of images into the image link. A new image can be injected from file or through a simple register interface. In Insertion mode the image link is blocked during injection whereas in Replacement mode the ImageInjector operator acts as a image sink, i.e. consumes all incoming images at any rate. |  |

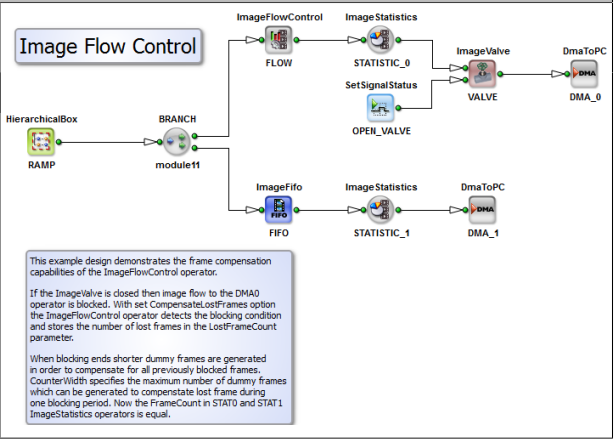
## 12.6.5. Image Monitoring

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DebuggingAndTest\ImageMonitor.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>In this example design the ImageMonitor operator allows the investigation of transfer state at a defined link position. |  |

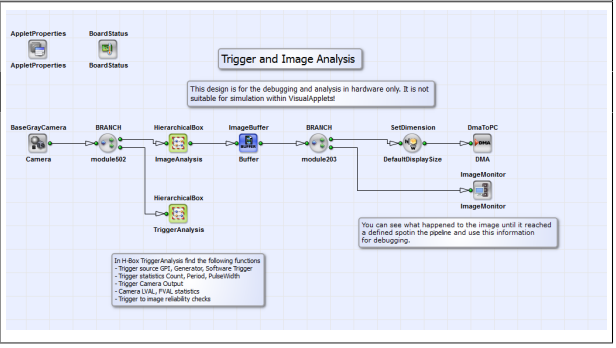
## 12.6.6. Image Grayscale Scope

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DebuggingAndTest\Scope.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>The Scope operator provides options for analyzing gray-scale pictures. The operator outputs a 2D waveform for each image channel by sampling input image lines. Up to four channels are supported. In a channel, each incoming line is sampled once. According to the settings you define in parameter SampleMode, the gray scale value of the first pixel in the line, the last pixel in the line, the smallest pixel value in the line, or the greatest pixel value in the line is used for sampling. |  |

## 12.6.7. Image Flow Control

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DebuggingAndTest\ImageFlowControl.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>This example design demonstrates the frame compensation capabilities of the ImageFlowControl operator. |  |

## 12.6.8. Trigger And Image Statistics

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DebuggingAndTest\TriggerAndImageStatistics.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Analysis of trigger and Camera Link functionalities is provided. The user can analyze the image data stream of a grayscale camera in Camera Link base configuration. The design is intended for the use in hardware only. | |

With the example design "TriggerAndImageStatistics.va" the analysis of trigger and Camera Link functionalities like number of pulses, signal width and period is possible. Also the difference between trigger and Camera Link signals is measured. The user can choose for the generation of the trigger signal between a software trigger, a trigger generator or an external source. In this design the user has furthermore the possibility to analyze the image data stream of a grayscale camera in Camera Link base configuration. The design is intended for the use in hardware only. It is not intended for the simulation within VisualApplets. This design is suitable to be a debugging "add-on" for image processing designs. In the following the functionalities of "TriggerAndImageStatistics.va" are explained. Its basic design structure is shown in Fig. 12.59.

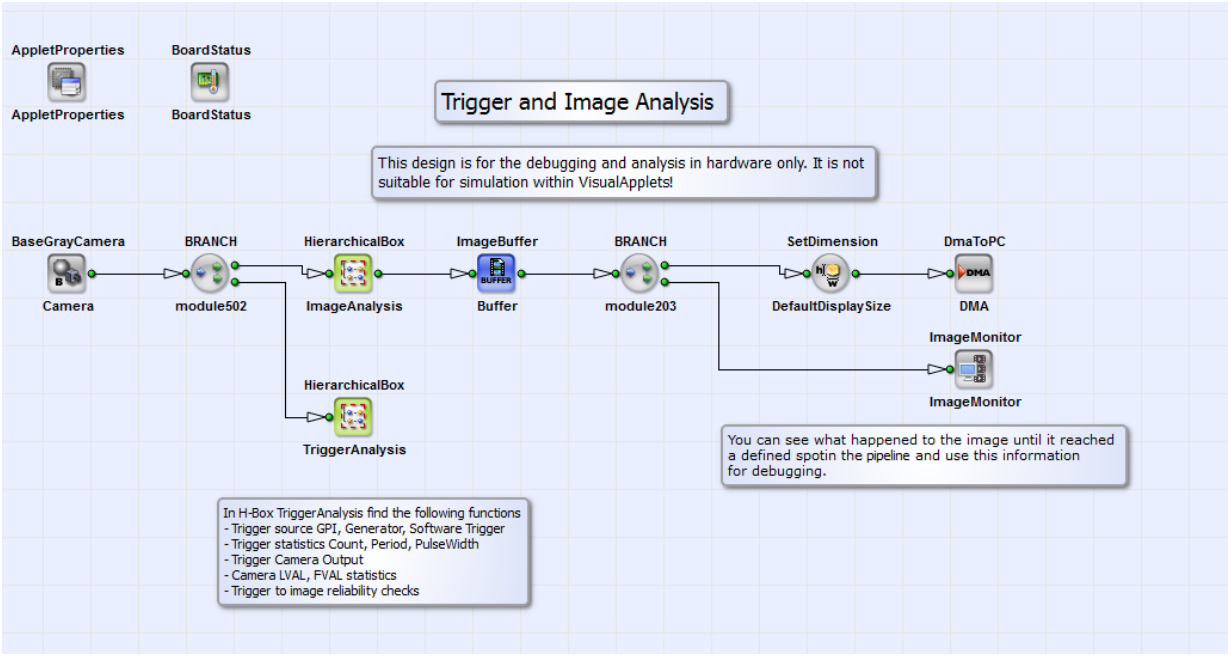



Figure 12.59. Basic design structure of the VA design "TriggerAndImageStatistics.va"

In the HierarchicalBox **ImageAnalysis** the image data stream of a grayscale camera in Camera Link base configuration is analyzed. Image properties of the current frame but also of a complete image sequence, like image dimensions and line length deviations between single frames, are measured there. The frames are sent to PC via DMA. Furthermore the user has the possibility to monitor the image at a any defined point in the pipeline using the debugging operator **ImageMonitor**. In the HierarchicalBox **TriggerStatistics** the properties of the trigger signal and also of Camera Link signals like LVAL and FVAL are measured. Please see Fig. 12.60 for the content of box **TriggerAnalysis**.

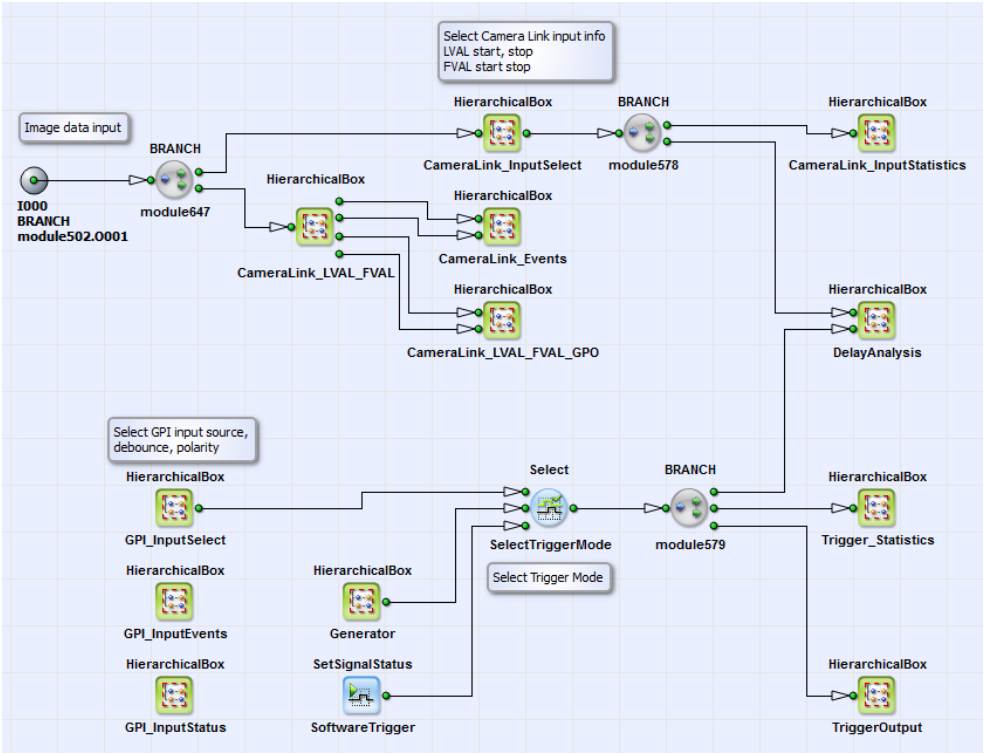



Figure 12.60. Content of HierarchicalBox **TriggerAnalysis**

Here in box **CameraLink_LVAL_FVAL** the LVAL and FVAL signals are defined. At beginning and end of each image line and each frame a signal is generated with operators **LineStartToSignal**/**LineEndToSignal** and **FrameStartToSignal**/**FrameEndToSignal**. The operator **RsFlipFlop** combines the corresponding start/stop signals to the LVAL and FVAL signals, which the user can monitor as events. These events are defined in the HierarchicalBox **CameraLink_Events** with the operator **EventToHost**. In the box **CameraLink_LVAl_FVAL_GPO** the LVAL and FVAL signals are assigned to the GPO and Front GPO with PinID 1. The statistics of the LVAL and FVAL start and stop pulses are defined in the HierarchicalBox **CameraLink_InputStatistics**. You can see the content of this box in Fig. 12.61.

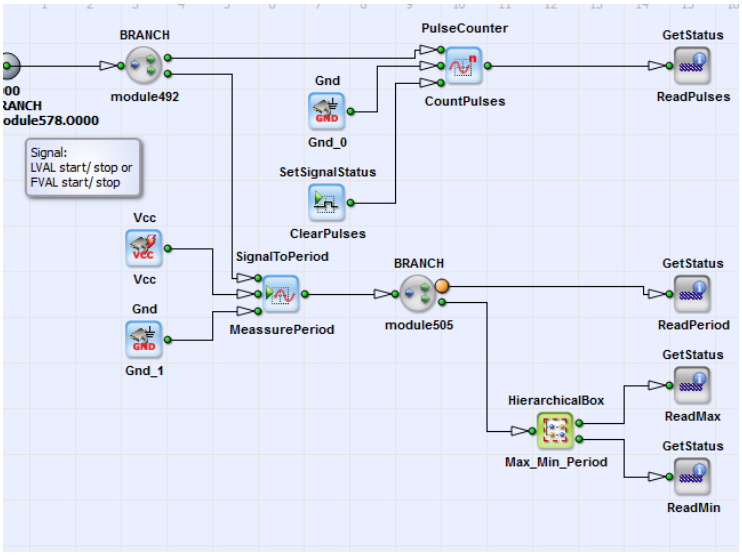



Figure 12.61. Content of HierarchicalBox **CameraLink_InputStatistics**

The number of pulses is here counted with operator **PulseCounter**. The user can monitor this number via the operator **GetStatus_ReadPulses**. The period between LVAL or FVAL start or stop

pulses is defined with operator **SignalToPeriod**. The user can monitor this value via operator **GetStatus_ReadPeriod** together with its maximum and minimum values via **GetStatus_ReadMax** and **GetStatus_ReadMin**. Beside the monitoring of the statistics of the Camera Link signal, the user can observe the statistics of the trigger signal. The trigger signal can be generated using a trigger **Generator**, a **SoftwareTrigger** or external trigger source via GPIs, which can be selected in **GPI_InputSelect**. The user can monitor the GPI input events and the input status (see boxes **GPI_InputEvents** and **GPI_InputStatus**). The trigger signal is applied to the camera control and GPOs in HierarchicalBoax **TriggerOutput**. The user can monitor the statistics of the trigger signal as defined in the HierarchicalBox **Trigger_Statistics**. You can see its content in Fig. 12.62.

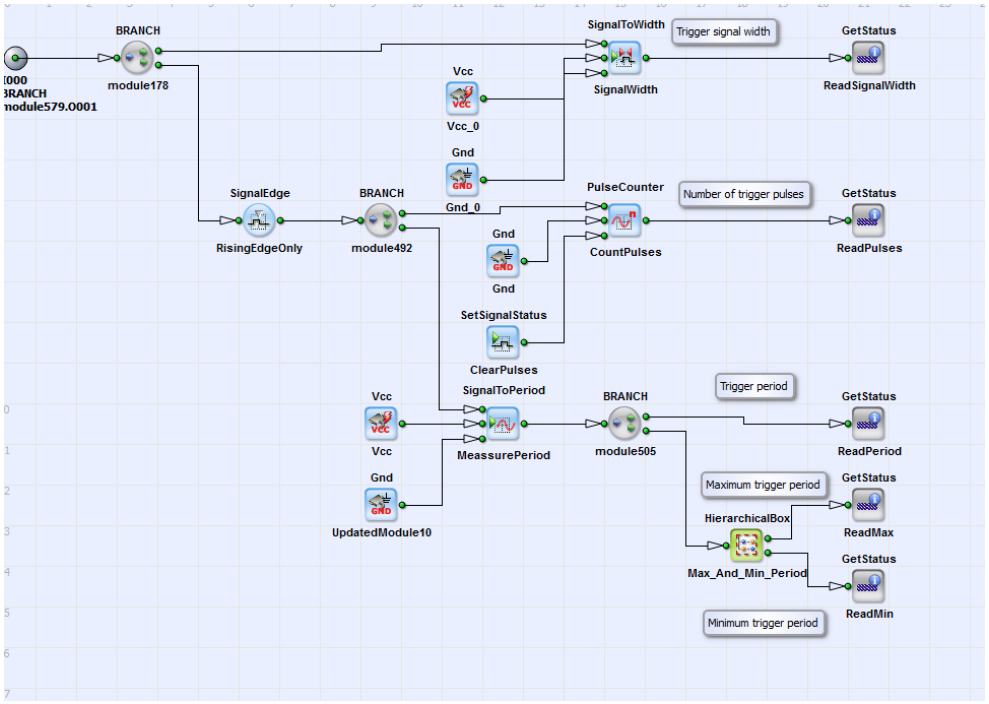



Figure 12.62. Content of HierarchicalBox **Trigger_Statistics**

Here the trigger signal width is defined (see **SignalToWidth** operator **SignalWidth**). The user can observe this value via **GetStatus** operator **ReadSignalWidth**. The number of trigger rising edge pulses is counted with **PulseCounter**. The user can have a look at this number via **GetStatus** operator **ReadPulses**. This is also true for the period between the trigger rising edge pulses, its maximum and minimum value (see **GetStatus** operators **ReadPeriod**, **ReadMax**, **ReadMin**). Coming back to Fig. 12.60. In the HierarchicalBox **DelayAnalysis** the CameraLink signals and the trigger signal are compared. You can see the content of **DelayAnalysis** in Fig. 12.63.

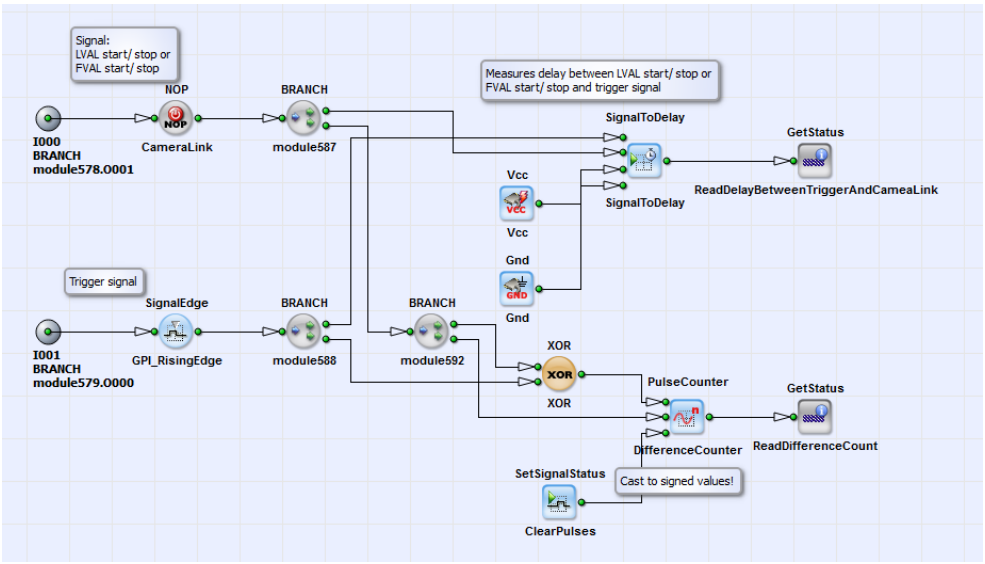



Figure 12.63. Content of HierarchicalBox **DelayAnalysis**

Here the delay between LVAL or FVAL start or stop signal and the trigger rising edge is measured. The user has access to this value via a **GetStatus** operator. Also the difference number of pulses between between the two signal is measured. If the user observes a value of **ReadDifferenceCount** greater than one, the trigger signals have been sent but the camera did not respond. If this value is smaller than 1 the camera sent more frames or lines than trigger pulses came in. Beside the analysis of trigger and Camera Link signals, the analysis of image data stream is implemented in "TriggerAndImageStatistics.va" in the HierachicalBox **ImageAnalysis** (see Fig. 12.59). You can see its content in Fig. 12.64.

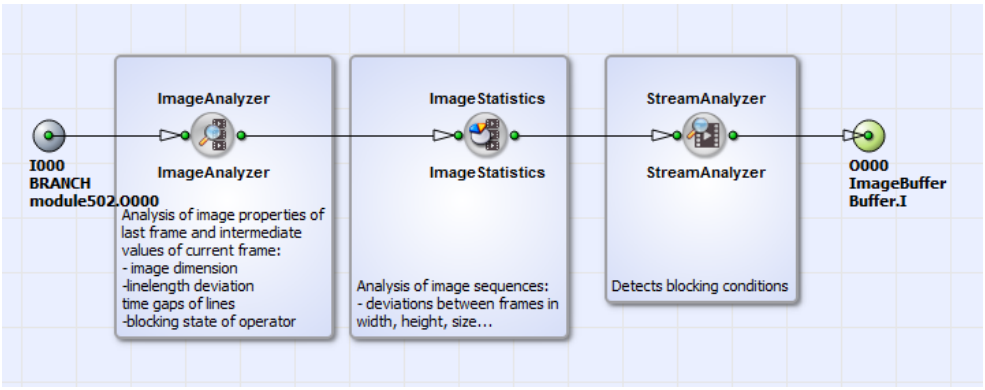


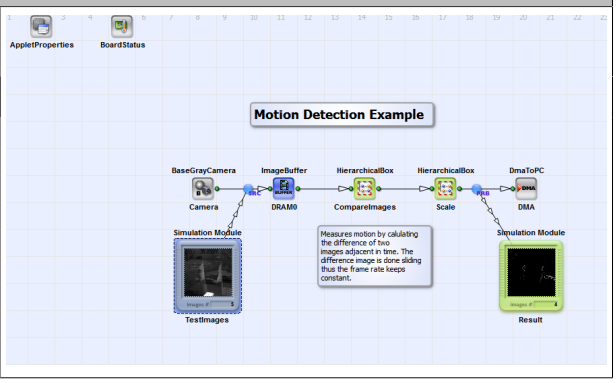

Figure 12.64. Content of HierarchicalBox **ImageAnalysis**

The operator **ImageAnalyzer** gives you the possibility to analyze the image properties of the last frame and the intermediate values of the current frame. These properties are image dimensions, line length deviations and time gaps of lines. This operator gives you also information on the blocking state of the operator input. The operator **ImageStatistics** analyzes complete image sequences. Deviations in image dimensions line gaps etc. are measured. The operator **StreamAnalyzer** gives you information on the data flow and blocking conditions. During the analysis the image data are not touched.

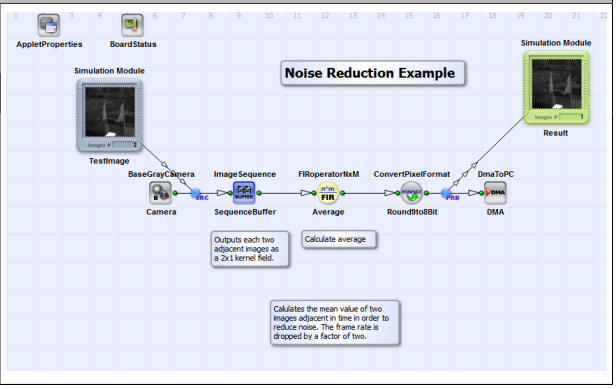
## 12.7. Difference Images

In this section you find two examples using the difference of two images for noise reduction and motion detection.

### 12.7.1. Motion Detection

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DifferenceImage \MotionDetection\MotionDetection.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Calculates the differences between two successive images. The differences are thresholded and output via DMA channel. |  |

### 12.7.2. Noise Reduction

| Brief Description | |
|---|---|
| **File:** \examples\Processing\DifferenceImage \NoiseReduction\NoiseReduction.va<br>**Default Platform:** mE4-VD4-CL<br>**Short Description**<br>The average of two acquired images is calculated to reduce noise. |  |

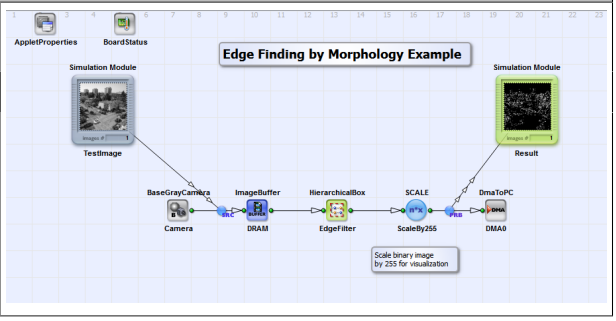
## 12.8. Filter

The following subsections contain examples for different kinds of filters. These filters are for edge detection, morphology, noise reduction and sharpening. Also general filter principles are introduced.

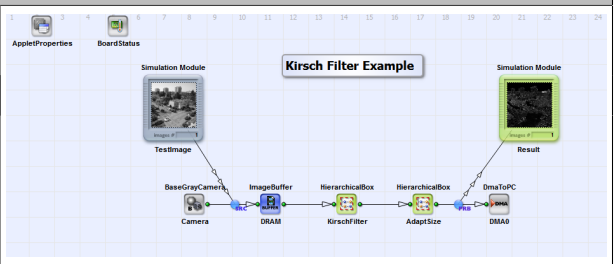
### 12.8.1. Edge Detection

This subsection contains examples on edge detection filters such as Sobel filter, Kirsch filter, Roberts Cross Gradient filter and an example about edge finding by morphology.

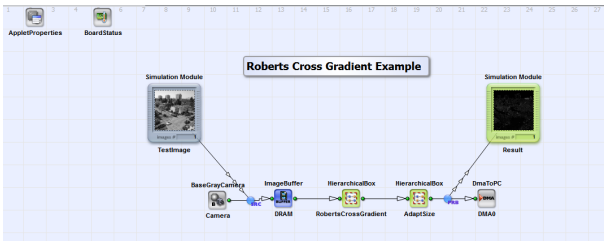
#### 12.8.1.1. Morphological Edge

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\EdgeDetection \MorphologicalEdge\MorphologicalEdge.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Edge detection. For every difference between the image and the eroded image, an edge is assumed. |  |

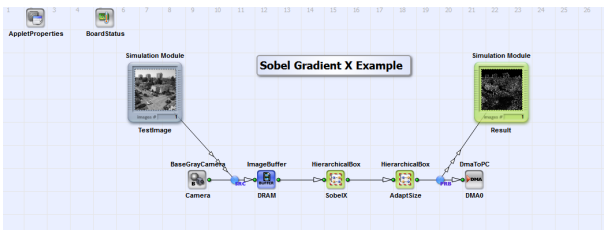
### 12.8.1.2. Kirsch Filter

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\EdgeDetection\KirschFilter\KirschFilter.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>The Kirsch filter is a good edge detection filter for non directional edges. |  |

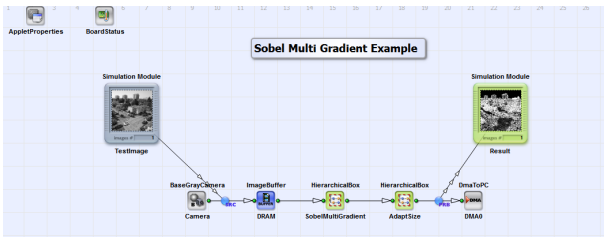
### 12.8.1.3. Roberts Cross Gradient

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\EdgeDetection\Roberts_Cross_Gradient\Roberts_Cross_Gradient.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Roberts Cross Gradient filter example. |  |

### 12.8.1.4. Sobel Gradient X

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\EdgeDetection\Sobel_Gradient_X\Sobel_Gradient_X.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>A Sobel filter in x-direction only. |  |

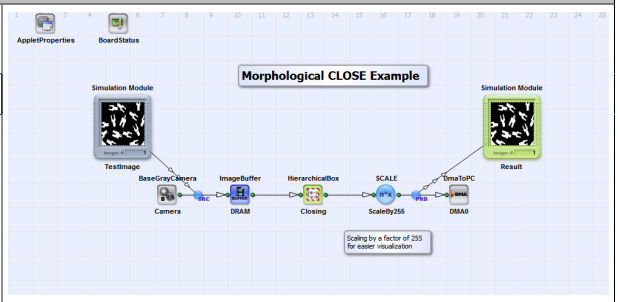
### 12.8.1.5. Sobel Multi Gradient

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\EdgeDetection\Sobel_Multi_Gradient\Sobel_Multi_Gradient.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>A Sobel filter in all 4 directions. |  |

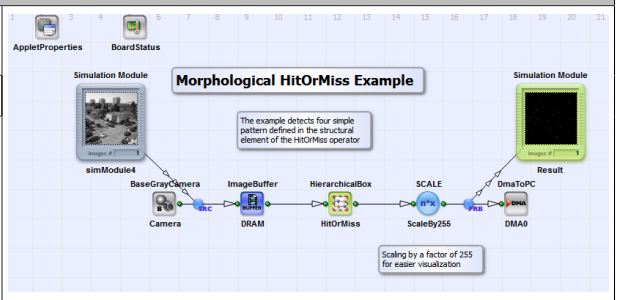
## 12.8.2. Morphology

You find in this subsection example designs on opening and closing and morphology edge detection. One example demonstrates how to find four simple patterns using the operator **HitOrMiss**.

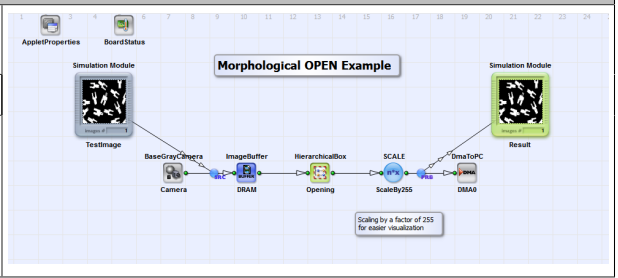
### 12.8.2.1. Close

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\Morphology \Close\Close.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows the implementation of a morphological close applied to binary images. | |

### 12.8.2.2. Hit or Miss

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\Morphology \HitOrMiss\HitOrMiss.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>The implementation can detect four simple patterns in a binary image. For every match, the output will be set to one. | |

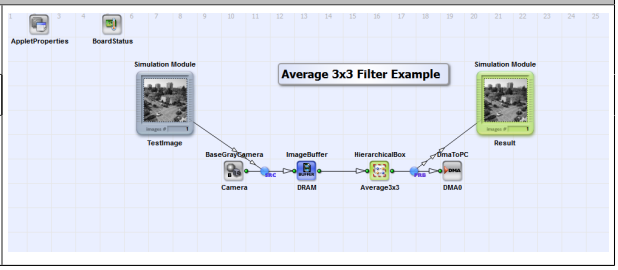
### 12.8.2.3. Open

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\Morphology \Open\Open.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows the implementation of a morphological open applied to binary images. | |

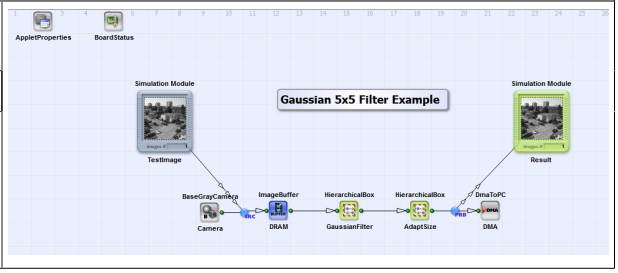
## 12.8.3. Noise Reduction

The examples in the following demonstrate how to reduce noise in an image. Three examples are provided using an average, a Gaussian and a median filter.

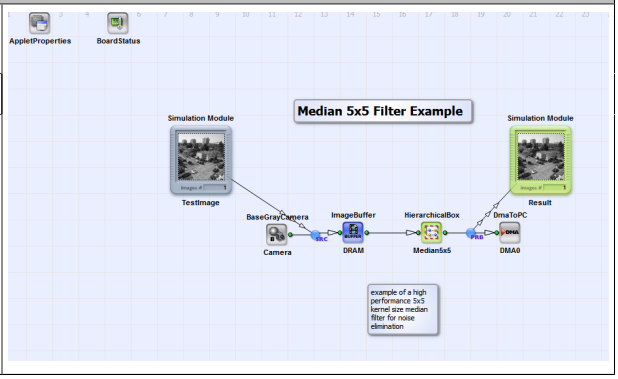
### 12.8.3.1. Averaging 3x3

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\NoiseReduction\Average3x3\Average3x3.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A simple 3x3 box filter. | |

### 12.8.3.2. Gaussian Filter 5x5

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\NoiseReduction\Gaussian5x5\Gaussian5x5.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A Gauss filter using a 5x5 kernel. | |

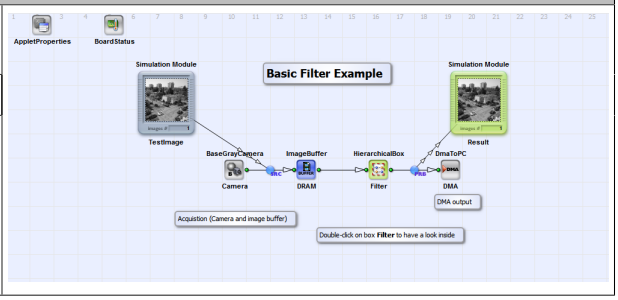
### 12.8.3.3. Median Filter 5x5

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\NoiseReduction\Median5x5\Median5x5.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Applet applies a 5x5 median filter on the image. | |

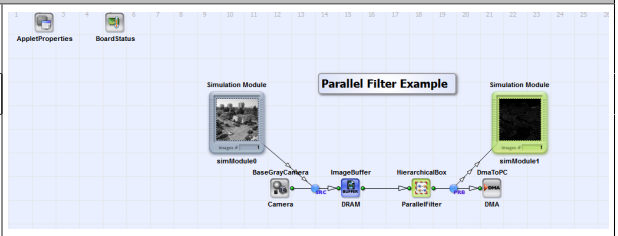
## 12.8.4. Principles

General principles about filtering are introduced in this subsection. It is demonstrated how you can implement and define a filter yourself, how to use a filter in parallel or for a line scan image.

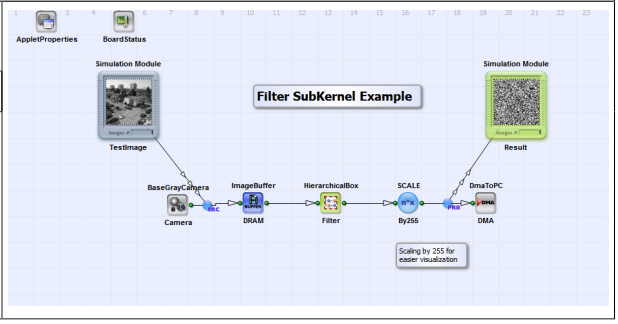
### 12.8.4.1. Filter Basics

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\Principles \FilterBasic\FilterBasic.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Explains the implementation of filters. Check the comments in the design file. | |

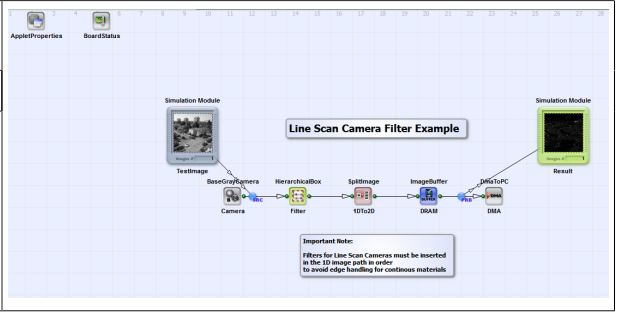
## 12.8.4.2. Parallel Filters

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\Principles \FilterParallel\FilterParallel.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>An example of the use of two filters in parallel. Check the synchronization rules in Section 3.6, 'Rules of Links', too. | |

## 12.8.4.3. Filter Sub Kernels

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\Principles \FilterSubKernels\FilterSubKernels.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows how to extract a sub kernel from a filter to obtain the original image data. This example performs a simple local adaptive binarization. See also Section 12.2.1, 'Adaptive Threshold'. | |

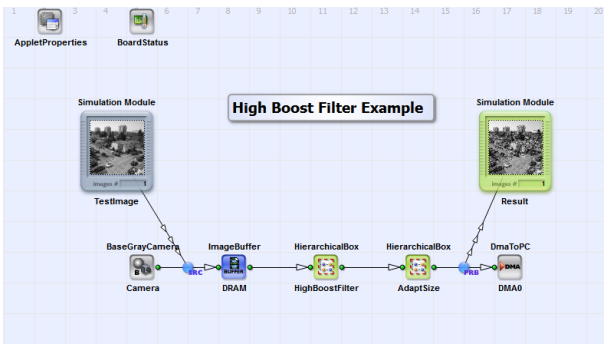
## 12.8.4.4. Filter for Line Scan Cameras

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Filter\Principles \FilterLineScan\FilterLineScan.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Explains how to implement a filter for line scan cameras. Check the comments in the design file. | |

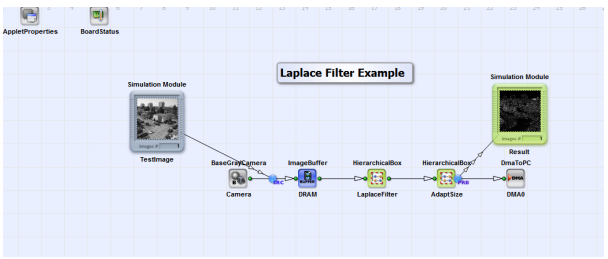
### 12.8.5. Sharpening

Two examples for sharpening of an image are described in this subsections. An HighBoost filter and a Laplace filter are implemented.

#### 12.8.5.1. High Boost Sharpening Filter

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Filter\Sharpening \HighBoost\HighBoost.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A high boost Laplace filter for sharpening. | |

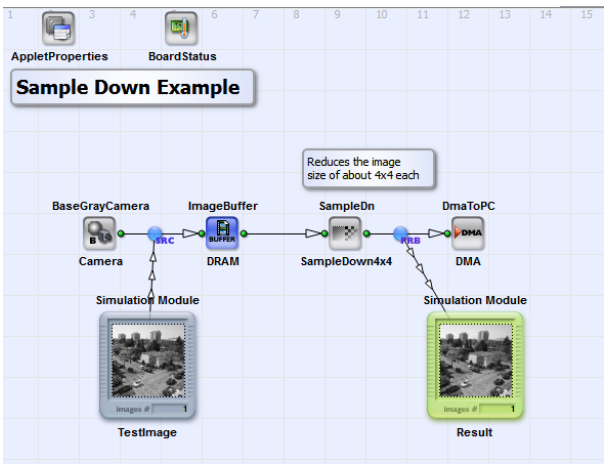
#### 12.8.5.2. Laplace Filter 3x3

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Filter\Sharpening \LaPlace3x3\LaPlace3x3.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A 3x3 Laplace filter. | |

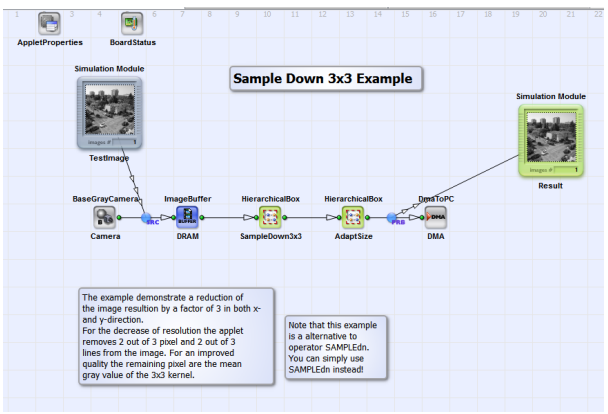
## 12.9. Geometry

In this section you find the description of examples on different kinds of geometry operations. These are example designs for down sampling of an image, the split and merge of an image and the scaling and shearing of a line scan image. Furthermore the mirroring and the calculation of the image moments of an object are performed.

### 12.9.1. Downsampling

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Geometry \Downsampling\Downsampling.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>The input image is downsampled i.e. reduced in size by 4x4. |  |

## 12.9.2. Downsampling 3x3

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Geometry \Downsampling3x3\Downsampling3x3.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Compared to the downsampling by 4x4 shown in Section 12.9.1, 'Downsampling' a downsampling by a factor of 3 by 3 is presented. The example will not use the *SampleDn* operator. The downsampling is made using discrete operators. Using *SampleDn* is possible, too. |  |

## 12.9.3. Geometric Transformation and Distortion Correction

We have implemented examples performing geometric transformation, geometric transformation in combination with distortion and Keystone correction and geometric transformation with image moments. In the following sections we first describe the mathematical background and then we introduce the VisualApplets designs.

| Example | Description |
|---|---|
| **"GeometricTransformation_ FrameBufferRandomRead.va"** | Geometric transformation:translation, rotation, scaling using operator **FrameBufferRandomRead** |
| **"GeometricTransformation_ ImageMoments.va"** | Geometric transformation:translation, rotation, scaling using image moments and **FrameBufferRandomRead** |
| **"GeometricTransformation_ PixelReplicator.va"** | Geometric transformation:translation, rotation, scaling using operator **PixelReplicator** |
| **"GeometricTransformation_ DistortionCorrection.va"** | Geometric transformation:translation, rotation, scaling, distortion and Keystone correction using operator **PixelReplicator** |

| Example | Description |
|---|---|
| **"DistortionCorrection.va"** | Distortion correction using operator **PixelReplicator** |

Table 12.5. List of Geometric Transformation Examples

## 12.9.3.1. Theoretical Background

### 12.9.3.1.1. Geometric Transformation

A combination of image rotation and translation in 2 dimensions with coordinates x and y can be expressed with:

$$\begin{pmatrix} x' \\ y' \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & T_x \\ sin(\phi) & \cos(\phi) & T_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{12.19}$$

Here x' and y' are the coordinates in the target image. The rotation angle is $\phi$ and $T_x$, $T_y$ are the translation parameters in x and y direction. The inverse translation and rotation is then:

$$\begin{pmatrix} x \\ y \\ 0 \end{pmatrix} = \begin{pmatrix} \cos(\phi) & \sin(\phi) & -\sin(\phi) \cdot T_y - \cos(\phi) \cdot T_x \\ -sin(\phi) & \cos(\phi) & \sin(\phi) \cdot T_x - \cos(\phi) \cdot T_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} . \tag{12.20}$$

And therefore the coordinates in the source image are calculated as:

$$\begin{aligned} x &= \cos(\phi) \cdot x' + \sin(\phi) \cdot y' - \sin(\phi) \cdot T_y - \cos(\phi) \cdot T_x \\ &\text{and} \\ y &= -\sin(\phi) \cdot x' + \cos(\phi) \cdot y' + \sin(\phi) \cdot T_x - \cos(\phi) \cdot T_y . \end{aligned} \tag{12.21}$$

For a VA design we need the inverse transformation for coordinate calculation due to target-to-source mapping. Scaling of the coordinates can be realized by multiplication with constants $S_x$, $S_y$ for x and y direction.

### 12.9.3.1.2. Distortion Correction

Optical distortion appears when the magnification of an object changes with distant to optical axis. The image has then barrel or pincushion shape. For the VA design we use a polynomial ansatz according to [Par09]. The distorted coordinates $x_d, y_d$ are calculated from undistorted coordinates $x_u, y_u$ as:

$$\begin{aligned} &\begin{pmatrix} x_d \\ y_d \end{pmatrix} = (1 + r_u^2 \cdot k_1 + r_u^4 \cdot k_2) \cdot \begin{pmatrix} x_u \\ y_u \end{pmatrix} = C(r_u) \cdot \begin{pmatrix} x_u \\ y_u \end{pmatrix} . \\ &with \\ &r_u = \sqrt{x_u^2 + y_u^2} . \end{aligned} \tag{12.22}$$

The coordinates are relative to the optical center of distortion. The parameters $k_1$, $k_2$ are the distortion coefficients. Some open source programs e.g. OpenCV give the possibility to calculate those parameters with calibration images [Ope16a]. For all $r_u$ a lookup-table with the correction parameters $C(r_u)$ can be calculated with a Matlab program. You can find it under \examples\Processing\Geometry \GeometricTransformation\LUTDistortionCorrection.m. With this table all distorted coordinates can be calculated by simple multiplication with the undistorted coordinates.

#### 12.9.3.1.3. Keystone Correction

This trapezoidal distortion appears when the camera is not placed perpendicular to the object to be filmed. With a 3x3 transformation matrix M with elements $m_1$ to $m_9$ this effect can be corrected as [Ope16b]:

$$\begin{pmatrix} t \cdot x_{Kd} \\ t \cdot y_{Kd} \\ t \end{pmatrix} = \begin{pmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 \\ m_7 & m_8 & m_9 \end{pmatrix} \cdot \begin{pmatrix} x_{Kc} \\ y_{Kc} \\ 1 \end{pmatrix} . \tag{12.23}$$

Please notice that we perform here inverse correction due to target-to-source mapping. The variables $x_{Kd}$, $y_{Kd}$ and $x_{Kc}$, $y_{Kc}$ are the Keystone distorted and Keystone corrected coordinates respectively. $m_1$ to $m_9$ can be calculated with OpenCV functions [Ope16b].

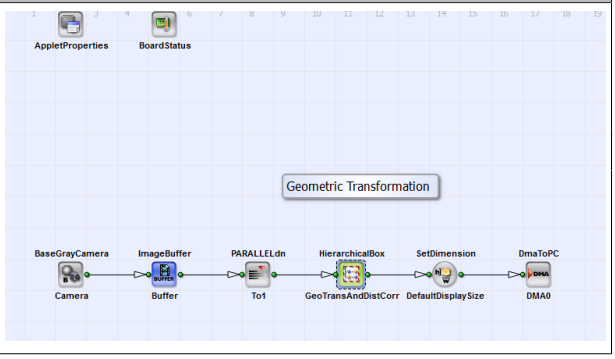
#### 12.9.3.1.4. Image Moments

Please see section 12.9.5 for detailed theoretical description of image moments.

### 12.9.3.2. Implementation in VisualApplets

In VisualApplets example designs are implemented performing geometric transformation, combined geometric transformation and distortion correction and geometric transformation controlled by image moments. The geometric transformation is implemented with target-to-source mapping i.e. inverse. This method is essential to guarantee, that there are no missing pixels in the target image sent to PC [Bur06]. We have implemented inverse geometric transformation on the basis of the operator **FrameBufferRandomRead** and alternative on the basis of the operator **PixelReplicator**. The differences will be explained in the following subsections. In these subsections the example designs are introduced in detail.

#### 12.9.3.2.1. Geometric Transformation Using FrameBufferRandomRead

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Geometry\GeometricTransformation \GeometricTransformation_ FrameBufferRandomRead.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Geometric Transformation: rotation, translation, scaling using the operator **FrameBufferRandomRead** | |

A geometric transformation is implemented in the VA design "Geometric Transformation_FrameBufferRandomRead.va" with target-to-source mapping. This method is essential to guarantee, that there are no missing pixels in the target image sent to PC [Bur06]. The coordinates of the source image $x$, $y$ are calculated with inverse geometric transformation for all target coordinates $x'$,$y'$. At the position of the coordinates $x$, $y$ the pixel value in the source image are read with the operator **FrameBufferRandomRead** at a parallelism of 1. The operator stores each pixel individually in DRAM and reads one pixel after each other. Linear interpolation helps to correct pixel values in the output image to PC when the calculated source image coordinates do not match pixel coordinates in the source image. In Fig. 12.65 you can see the basic design structure.

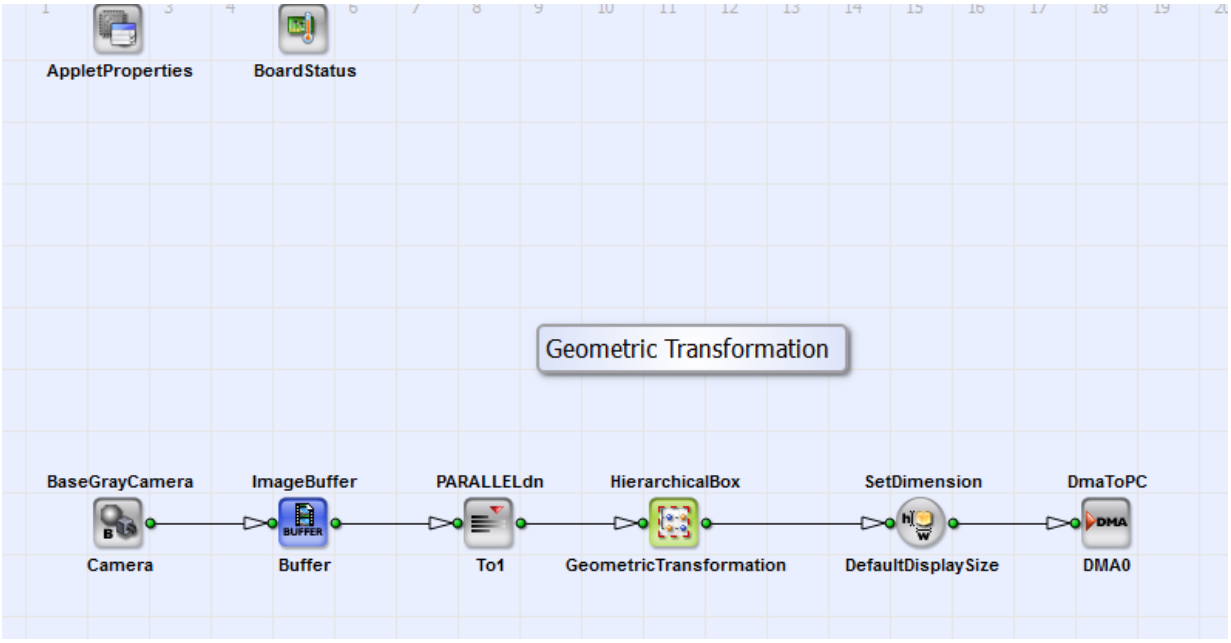



Figure 12.65. Basic design structure of the VA design "GeometricTransformation_FrameBufferRandomRead.va"

The design consists of a camera interface in Camera Link base configuration (**BaseGrayCamera_Camera**) and the HierarchicalBox **Geometric Transformation** in which the scaling, rotation and translation of the acquired images is performed. The operator **ImageBuffer_Buffer** gives the opportunity to store the image. Via DMA (operator **DmaToPC_DMA0**) the image is transferred to PC.

In 12.66 you can see the content of the HierarchicalBox **GeometricTransformation**. The parameters **IntParamTranslator**, **FloatParamReference** and **IntParamReference** give you the possibility to set the relevant parameters of the geometric transformation (value of scaling, rotation, translation and image dimensions) very comfortably. The properties are image **Height** and **Width**, the number of pixels you want to shift the image in x and y direction (parameter: **Translatex** and **Translatey**) and the scale factor in x and y direction **Scalex** and **Scaley**. You can set the rotation angle in degrees with the parameter **Phi**. As alternative you can set these **Properties** via "Right-Mouse-Click" on the HierarchicalBox **GeometricTransformation**. You can see the content of this box in Fig. 12.66.

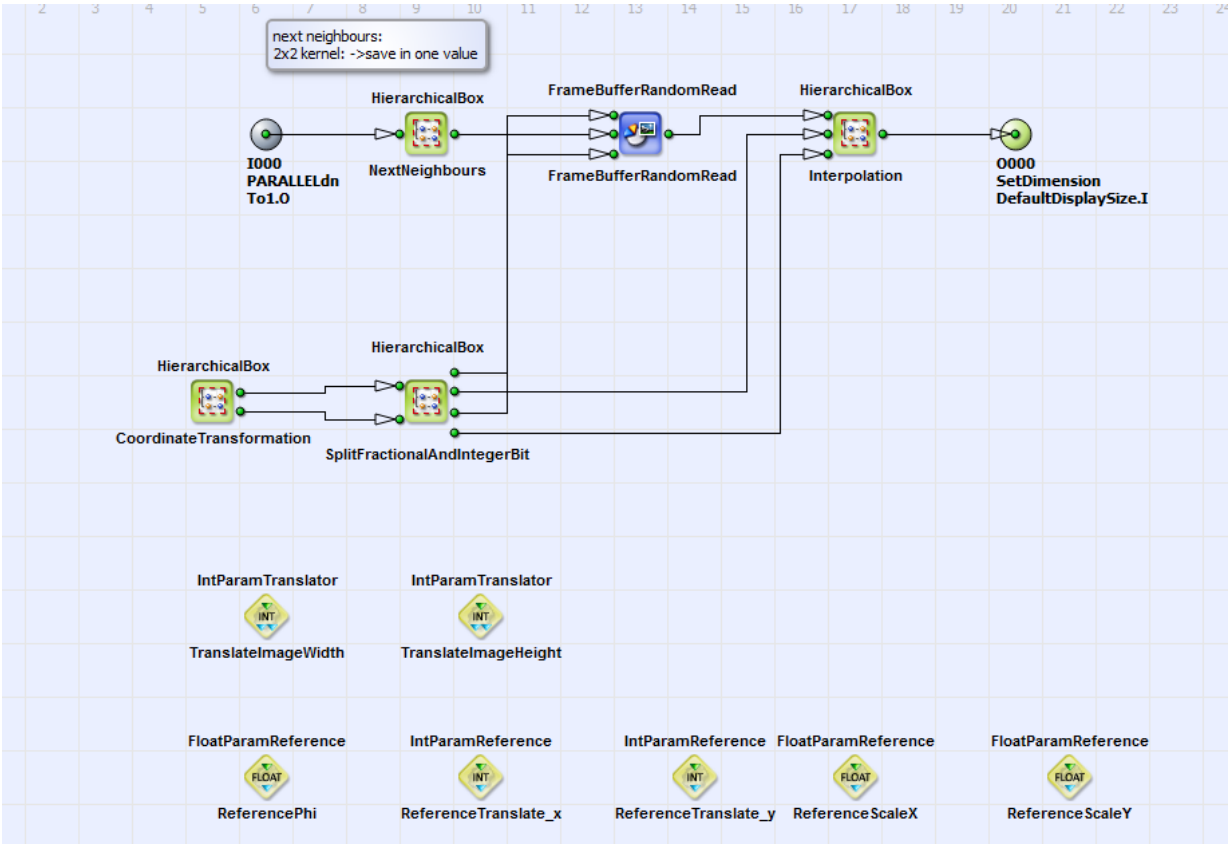



Figure 12.66. Content of HierarchicalBox"GeometricTransformation"

The **SourceImage** is linearly written to the buffer **FrameBufferRandomRead**. With each pixel the next neighbors are saved in a 2x2 kernel (HierarchicalBox **NextNeighbours**). The coordinates of the target image $x', y'$ (eq. 12.21) are created with the operators **CreateBlankImage**, **CoordinateX** and **CoordinateY** in the HierarchicalBox **OutputImage** (contained in HierarchicalBox **CoordinateTransformation**) (see Fig. 12.67 and 12.68).

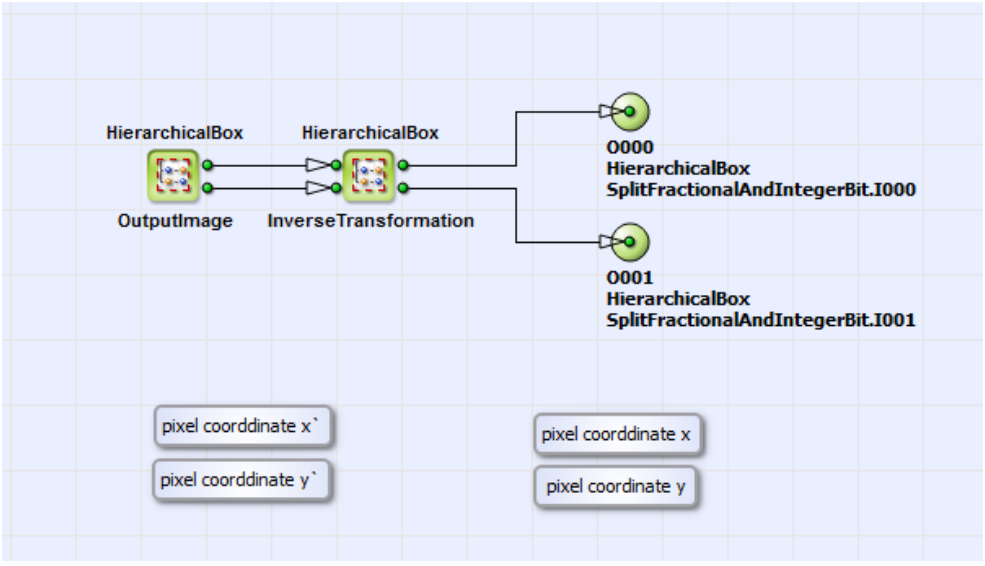



Figure 12.67. Content of HierarchicalBox **CoordinateTransformation**

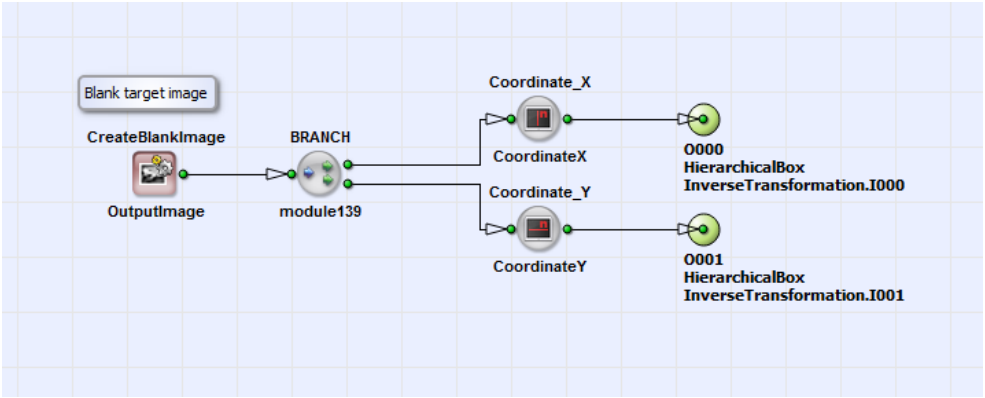



Figure 12.68. Content of HierarchicalBox **OutputImage**

The coordinates $x'$, $y'$ can be rotated, translated and/or scaled in inverse direction according to eq. 12.21. The rotation angle $\phi$ and the parameters $T_x$, $T_y$, $S_x$ and $S_y$ for translation and scaling can be chosen by setting the corresponding constants via the transport parameters as described above. In Fig. 12.69 you can see the VA implementation of eq. 12.21 with the possibility of additional scaling.

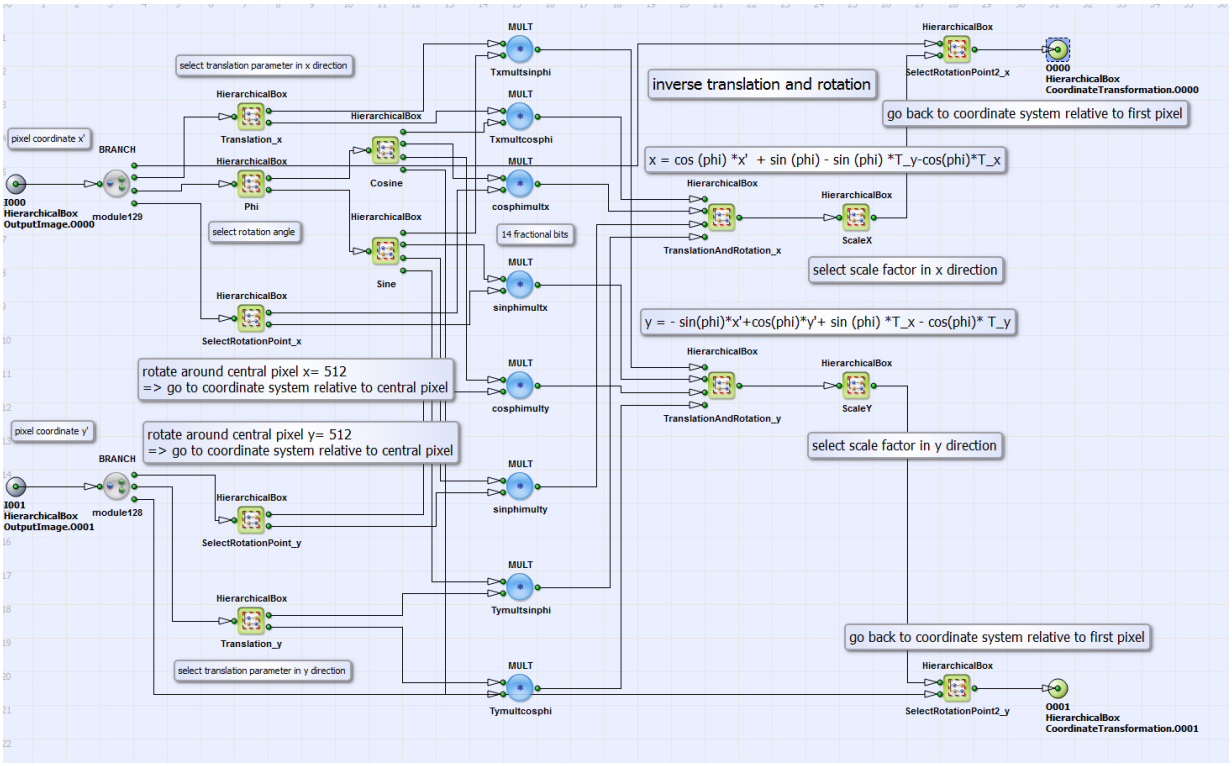



Figure 12.69. Content of HierarchicalBox **InverseTransformation**

The resulting coordinates of inverse transformation are the source coordinates $x$ and $y$ (eq. 12.21). In the HierarchicalBox **SplitFractionalAndIntegerBit** (see Fig. 12.66) the integer and fractional bit part of these coordinates are separated. The integer bit part, which corresponds to pixel positions in the source image, is the coordinate input for the operator **FrameBufferRandomRead** in x and y direction (see Fig. 12.66). This operator reads pixel value information (parallelism = 1) together with its next neighbors (see above) from the source image at the corresponding coordinates. The fractional bit part of the source coordinates $x$ and $y$ corresponds to interpixel positions in the source image. With these informations and the pixel value information from the neighboring pixels in the source image the true pixel value in the target image is bilinear interpolated in the HierarchicalBox **Interpolation**. Please see for detailed description for the interpolation process [Bur06].

In Fig. 12.70 and 12.71 you can see for an example geometric transformation the source and target images. A translation of 100 pixels in x and y direction and a rotation around 45° is performed.

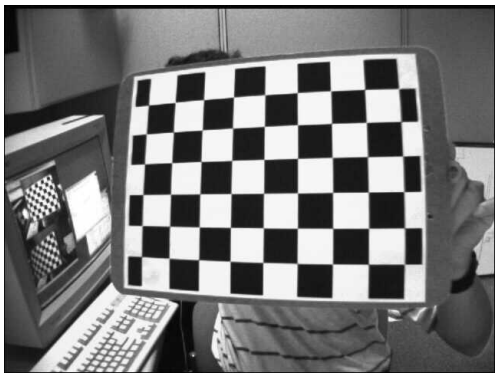


Figure 12.70. Source image [Ope16a]

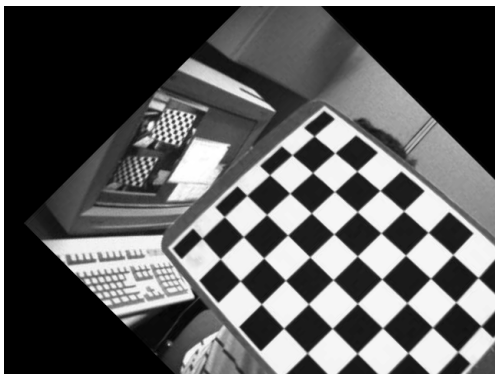
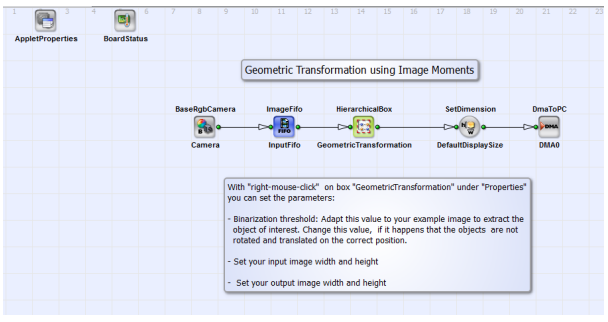


Figure 12.71. Rotated and translated target image

## 12.9.3.2.2. Geometric Transformation Using Image Moments

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Geometry\GeometricTransformation \GeometricTransformation_ImageMoments.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Geometric Transformation: rotation and translation of an object into the image center using image moments | |

In this example design the position and rotation angle of an object in a RGB image is determined with image moments. A rotation into a horizontal orientation and a translation into the image center is performed. As default the input image dimension is 640x240 pixels and the output image dimension 256x128 pixels. In Fig. 12.72 you can see the basic design structure.

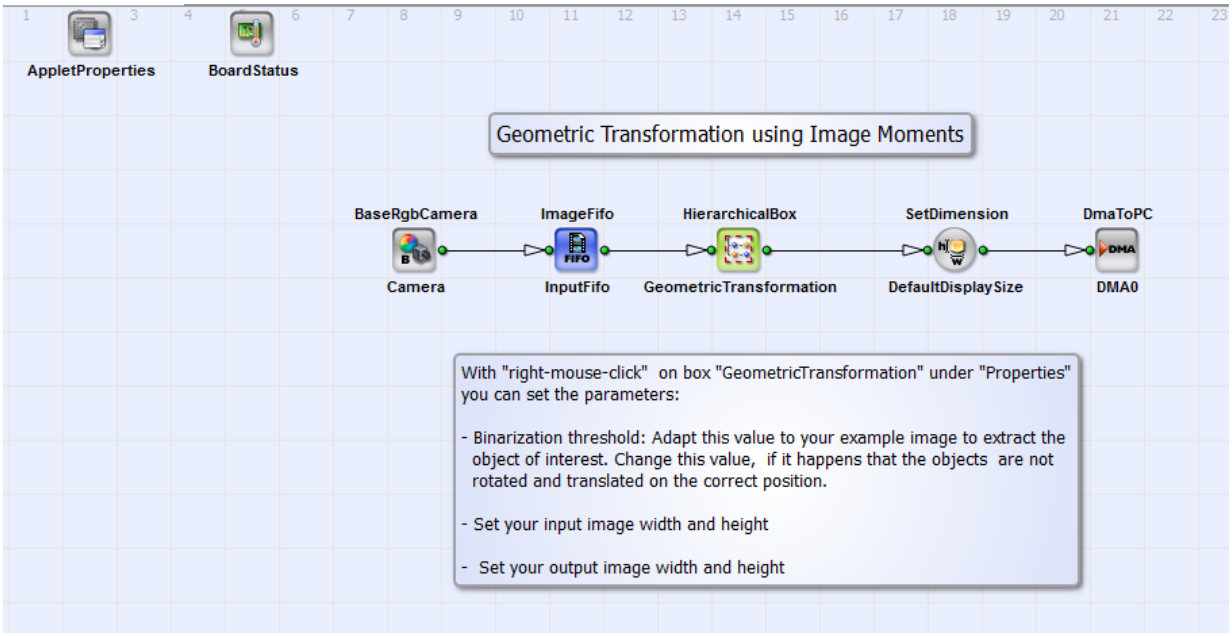



Figure 12.72. Basic design structure of the VA design "GeometricTransformation_ImageMoments.va"

The design consists of an RGB camera interface in Camera Link base configuration (**BaseRgbCamera_Camera**) and the HierarchicalBox **Geometric Transformation** in which the rotation and translation of the acquired images is performed. With "right-mouse-click" under "Properties" you can set the input and output image dimensions and the binarization threshold. This threshold is relevant for the calculation of the image moments as described below. The geometric transformation is done with target-to-source mapping (see introduction text in section 12.9.3.2). Via DMA (operator **DmaToPC_DMA0**) the image is transferred to PC. In Fig. 12.73 you can see the content of **Geometric Transformation**.

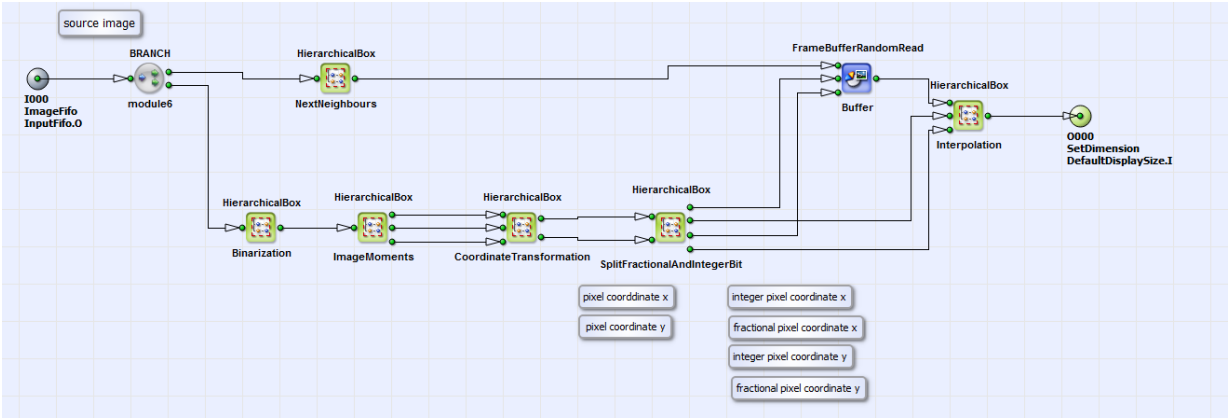



Figure 12.73. Content of the HierarchicalBox **GeometricTransformation**

The rotation angle relative to the horizontal position and the center of gravity position of the object is determined with image moments in the HierarchicalBox **ImageMoments** calculated on the binarized image (HierachicalBox **Binarization**). The content of **ImageMoments** is equivalent to the box **ImageMoments** of the VA example "ImageMoments.va" (see Fig. 12.93 in section 12.9.5) but without the calculation of the object eccentricity. The output links of the HierachicalBox **ImageMoments** (from up to down) are the center of gravity position in x and y direction and the rotation angle. The source coordinate calculation (according to eq. 12.21) based on these parameters is content of the HierachicalBox **CoordinateTransformation**. In 12.74 you can see the content of this box.

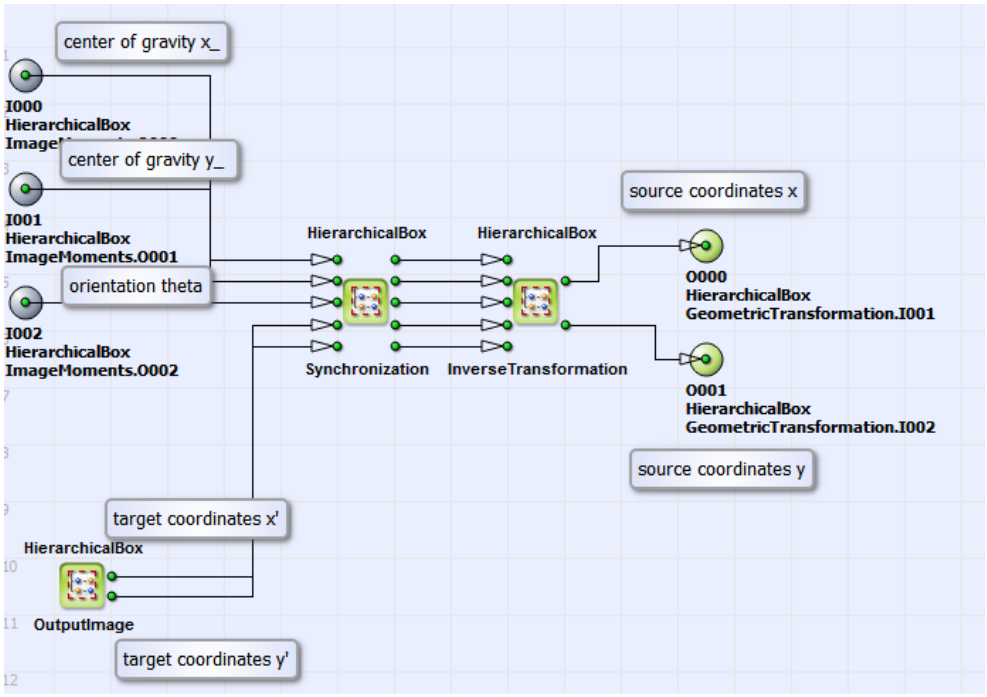



Figure 12.74. Content of HierarchicalBox **CoordinateTransformation**

In the HierarchicalBox **Synchronization** the single pixel values of center of gravity in x and y direction and rotation are extended to output image dimension, which is defined in the HierarchicalBox **OutputImage**. The output links of this box represent the target image coordinates $x', y'$. In the HierarchicalBox **InverseTransformation** the source coordinates $x$ and $y$ are calculated according to eq. 12.21. In Fig. 12.75 you can see the content of this box.

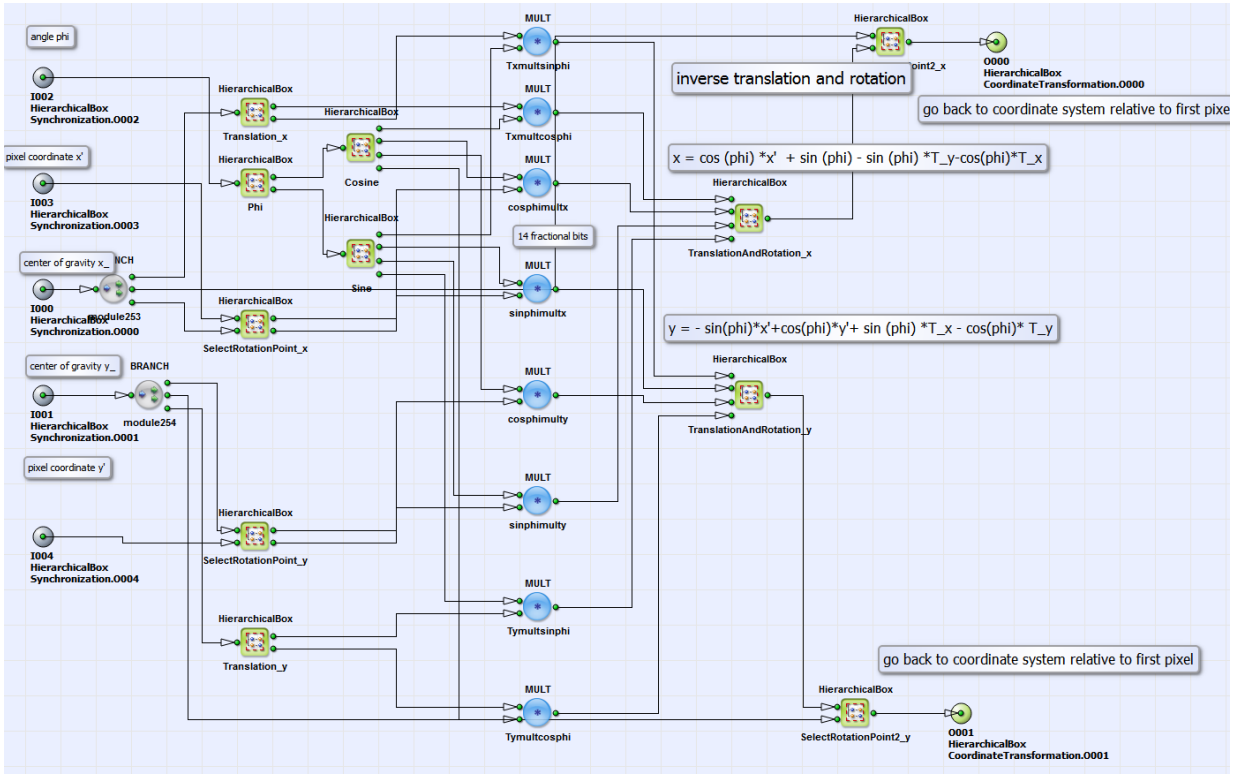



Figure 12.75. Content of HierarchicalBox **InverseTransformation**

The design structure in this box is similar to the implementation of inverse transformation in the examples "GeometricTransformation_FrameBufferRandomRead.va" and "GeometricTransformation_PixelReplicator.va". The difference is, that the translation parameter in x and y direction is determined by the difference between the center of gravity and the image center in x and y direction (see content of boxes **Translation_x** and **Translation_y**). The rotation angle (content of box **Phi**) is defined by the result of the image moments calculation. Coming back to the basic design structure in Fig. 12.72. The calculated source image coordinates $x$ and $y$ represent the output links of box **CoordinateTransformation**. In the box **GeometricTransformation** (see Fig. 12.76) the geometric transformation with the operator **FrameBufferRandomRead** and the bilinear interpolation of the correct target pixel value is performed analog to the example "GeometricTransformation_FrameBufferRandomRead.va" (see Fig. 12.66 in section 12.9.3.2.1).

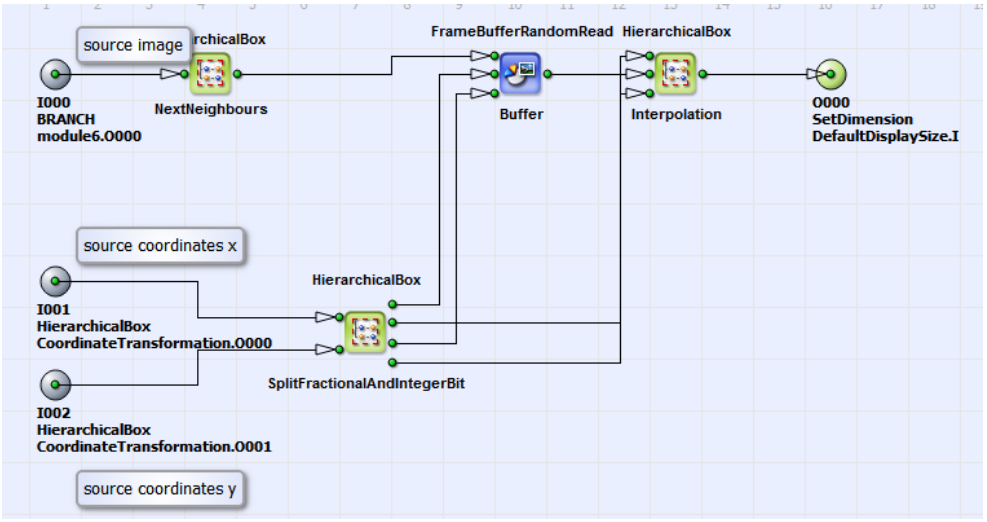



Figure 12.76. Content of HierarchicalBox **GeometricTransformation**

For an example object in Fig. 12.77 you can see the orientation and position corrected target image in Fig. 12.78.

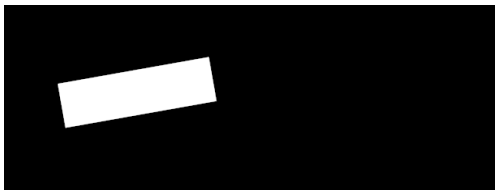
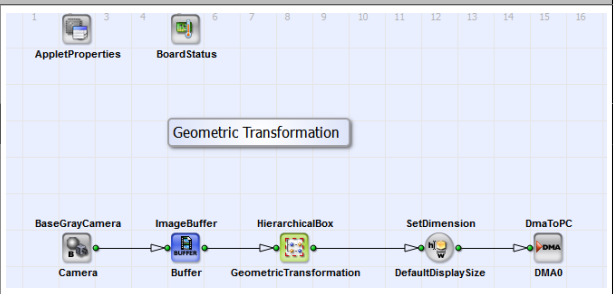


Figure 12.77. Source image (dimension: 640x240 pixels)



Figure 12.78. Position and orientation corrected target image (dimension: 256x128 pixels)

## 12.9.3.2.3. Geometric Transformation using PixelReplicator

| Brief Description | |
|---|---|
| **File:** \examples\Processing \Geometry\GeometricTransformation \GeometricTransformation_PixelReplicator.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Geometric Transformation: rotation, translation, scaling using the operator **PixelReplicator** | |

The VisualApplets design example "GeometricTransformation_PixelReplicator.va" performs the same geometric transformation as the example "GeometricTransformation_FrameBufferRandomRead.va" but with higher performance. As explained above the operator **FrameBufferRandomRead** stores each pixel individually in DRAM and reads them one after each other. In this example a block of a certain amount of pixels is stored in a DRAM cell. The example design reads only from DRAM if the pixel to be read is not in the same cell as the previous pixel. The example in Fig. 12.79 with 8 pixel per DRAM cell gives an idea of the performance increasement.

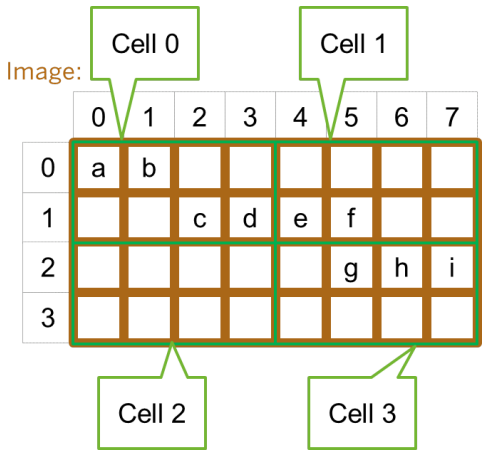



Figure 12.79. Example: 8 Pixels are stored in one DRAM cell

The DRAM blocks are marked in green. To read the 9 pixels a to i you need 3 DRAM cycles instead of 9. We achieve an effective parallelism of 3 here. The maximum amount of 8 bit pixel, which can be stored in one RAM cell, is 32 (marathon VCL data width: 256 Bit). The shape of the ROI for RAM cell block is defined by the rotation angle of the geometric transformation. As default a RAM cell shape of 9x3 pixels is chosen in this example. The basic design structure is equivalent to the design structure in Fig. 12.65 but without the operator **PARALLELdn_To1**. The content of the HierarchicalBox **GeometricTransformation** is also equivalent to the corresponding box (Fig. 12.66) in the example "GeometricTransformation_FrameBufferRandomRead.va". The difference is that the operator **FrameBufferRandomRead** is replaced by the HierarchicalBox **FrameBufferRandomRead_Par8**. You can see its content in Fig. 12.80.

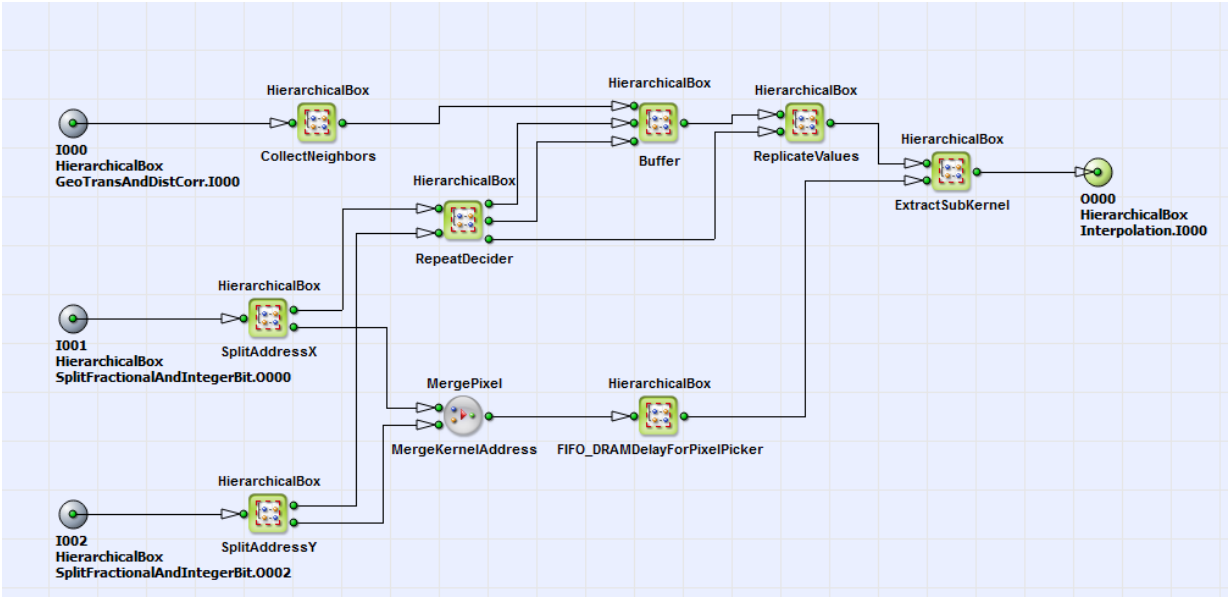
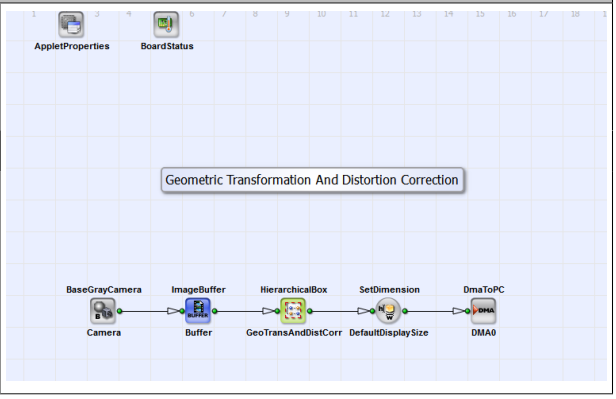



Figure 12.80. Content of HierarchicalBox **FrameBufferRandomRead_Par8**

In the box **CollectNeighbors** the source image is split into sub-ROIs, which define the size of each RAM cell. Here a RAM cell size of 9x3 is chosen. In the HierarchicalBoxes **SplitAdressX** and **SplitAdressY** the integer bit part of the source coordinates $x$ and $y$ as result of inverse geometric transformation (see section 12.9.3.2.1) is split in two components. The upper bit part (upper output link of **SplitAdressX** and **SplitAdressY**) represents the position of the DRAM cell, the lower bit part represents the pixel position in the DRAM cell. A DRAM cell is read at the DRAM cell coordinates in the HierarchicalBox **Buffer** (contains operator **FrameBufferRandomRead**) if it was not decided in the HerarchicalBox **RepeatDecider**to reuse previous DRAM cell. If it is decided to reuse the previous cell, the pixel values of the DRAM cell need to be replicated. This is performed in the HierarchicalBox **ReplicateValues** with the operator **PixelReplicator**. In the box **ExtractSubKernel** the required pixel value together with its next neighbors is extracted from DRAM cell using pixel coordinates as output of **SplitAdressX** and **SplitAdressY** (lower output link). The bilinear interpolation of the correct target pixel value is performed equivalent to the example in 12.9.3.2.1.

## 12.9.3.2.4. Geometric Transformation and Distortion Correction

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Geometry\GeometricTransformation \GeometricTransformation_ DistortionCorrection.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Geometric Transformation: rotation, translation, scaling, distortion and Keystone correction using the operator **PixelReplicator** | |



The VA example "GeometricTransformation_DistortionCorrection.va" is an extension of the design "GeometricTransformation_PixelReplicator.va". In addition distortion and Keystone correction (according to eq. 12.22 and eq. 12.23) is implemented in this design. In the following only the different parts in comparison to "GeometricTransformation_PixelReplicator.va" will be explained. In Fig. 12.81 you can see the basic design structure.

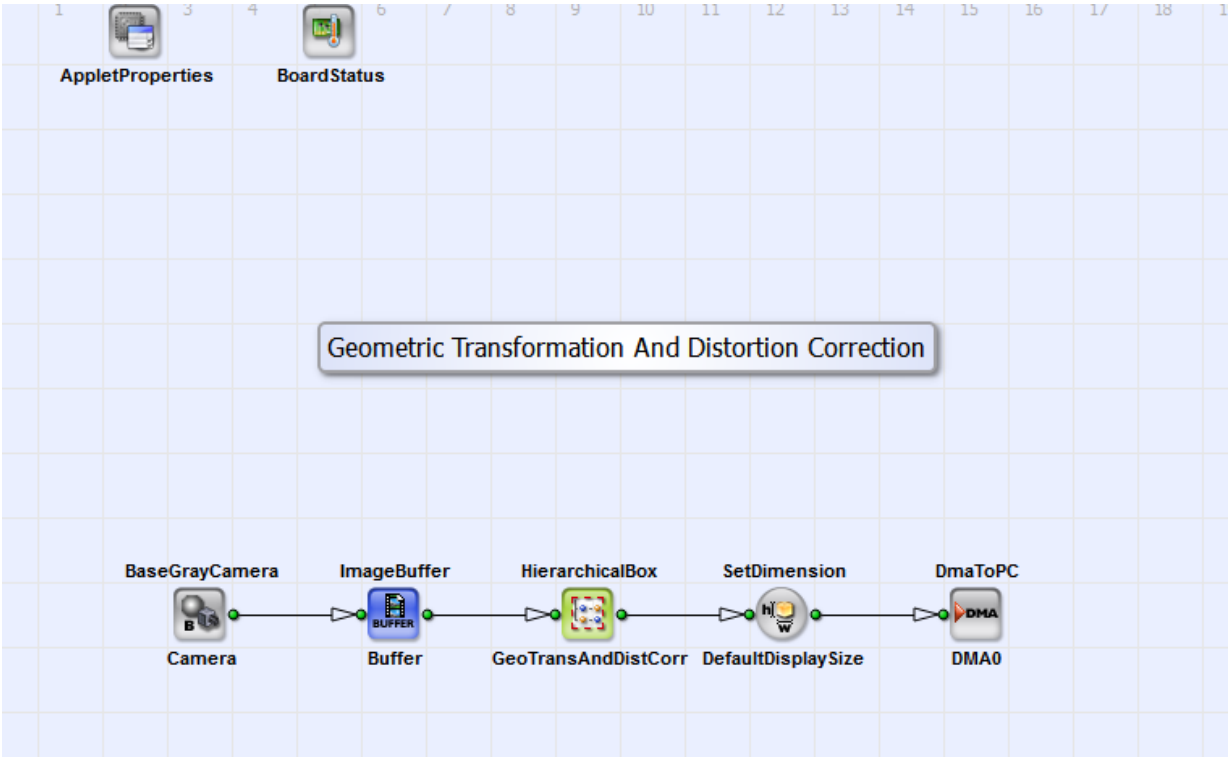



Figure 12.81. Basic design structure of the VA design "GeometricTransformation_DistortionCorrection.va"

A grayscale image (maximum dimension: 4096x4096 pixels) from a camera interface in CameraLink base configuration (**BaseGrayCamera_Camera**) is transformed in the HierarchicalBox **GeoTransAndDistCorr** and transferred to PC via DMA (**DmaToPC_DMA0**). The structure is equivalent to the design "GeometricTransformation_FrameBufferRandomRead" and "GeometricTransformation_PixelReplicator". The structure of **GeoTransAndDistCorr** is equivalent to the box **GeometricTransformation** of the same two designs (see Fig. 12.66). Here the inverse transformation and in addition distortion and Keystone correction is performed. The content of box **CoordinateTransformation** (in box **GeometricTransformation**) is shown in Fig. 12.82.

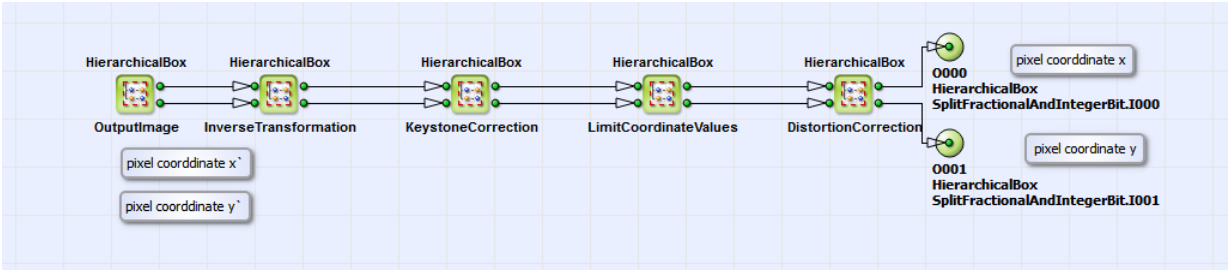



Figure 12.82. Content of HierarchicalBox **CoordinateTransformation**

Equivalent to the designs "GeometricTransformation_FrameBufferRandomRead" and "GeometricTransformation_PixelReplicator" (sections 12.9.3.2.1 and 12.9.3.2.3) the target coordinates $x', y'$ are inverse transformed according to eq. 12.21 with additional scaling (see also section 12.9.3.2.1). With the resulting coordinates of inverse transformation we perform then a inverse Keystone and distortion correction in the HierarchicalBoxes **KeystoneCorrrection** and **DistortionCorrection**. The content of these boxes are displayed in Fig. 12.83 and Fig. 12.82.

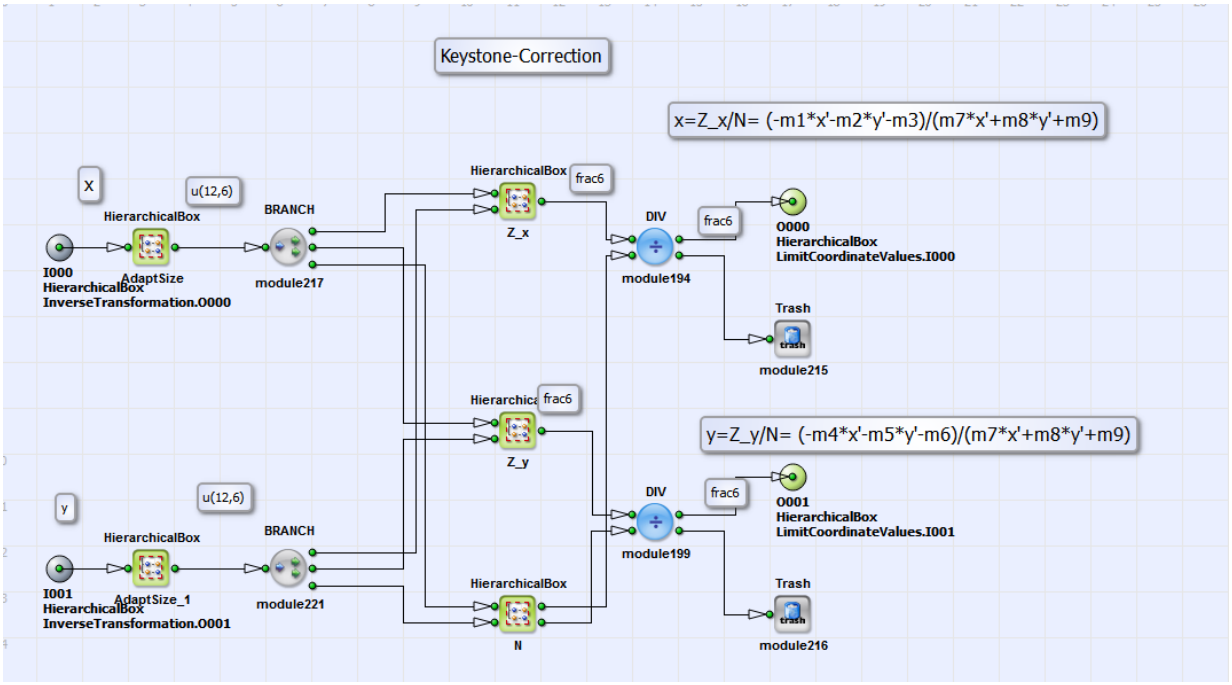



Figure 12.83. Content of HierarchicalBox **KeystoneCorrection**

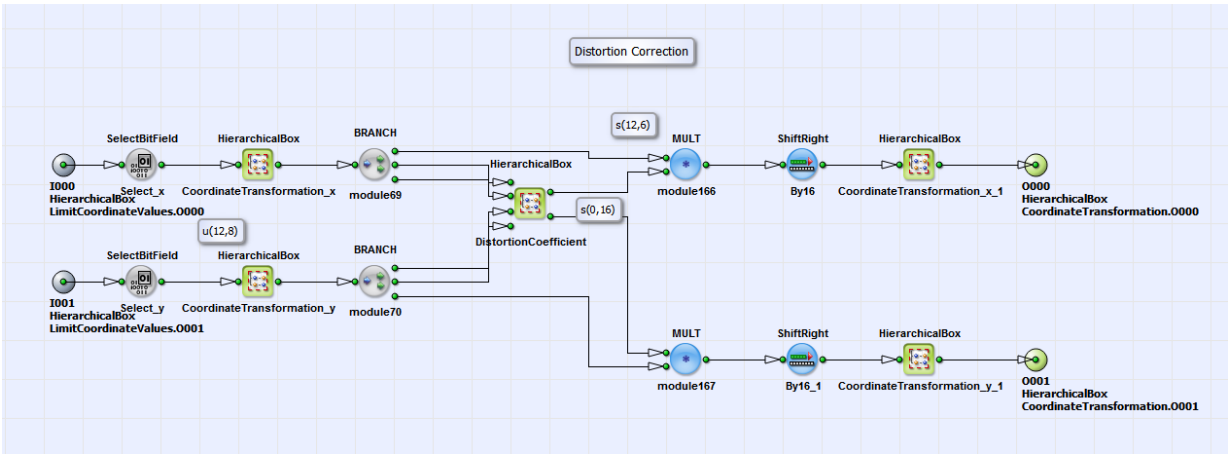



Figure 12.84. Content of HierarchicalBox **DistortionCorrection**

In these boxes eq. 12.22 and eq. 12.23 are implemented. The matrix elements m1 to m9 in the boxes **Z_x**, **Z_y** and **N** (see Fig. 12.83) can be calculated with the help of the OpenCV library [Ope16b]. For the example image "Example.tif" (under \examples\Processing\Geometry\GeometricTransformation) you find the example matrix values in the text file "MatrixValues.txt" at the same location. For the distortion correction in Fig. 12.82 first the coordinates are transformed in a coordinate system relative to the optical center (**CoordinateTransformation_x** and **CoordinateTransformation_y**), which is in most cases the image center. In the HierarchicalBox **DistortionCoefficient** the distance $r_u = \sqrt{x_u^2 + y_u^2}$ (see eq. 12.23) from optical center is calculated. For every $r_u$ then a distortion correction parameter $C(r_u)$ exists in a lookup table (see Fig. 12.85). The lookup-table is externally created with a Matlab module, which you can find under \examples\Processing\Geometry\GeometricTransformation \LUTDistortionCorrection.m and the distortion parameters $k_1$ and $k_2$ created with the OpenCV library [Ope16b]. In "LUTDistortionCorrection.m" you find as example the correction parameters $k_1$ and $k_2$ for the example image "Example.tif" (under \examples\Processing\Geometry\GeometricTransformation) [Ope16a]. The coefficient $C(r_u)$ is then multiplied with coordinates $x_u$ and $y_u$ (Fig. 12.82). The resulting distorted coordinates are then transformed back to a coordinate system relative to the "left upper image corner" (see boxes **CoordinateTransformation_x** and **CoordinateTransformation_y**). The result are the source image coordinates $x_d$ and $y_d$. The content of the HierarchicalBox

**LimitCoordinateValues** (Fig. 12.82) sets boundary conditions for the transformed image coordinates. After separation of integer and fractional bit parts (see box **SplitFractionalAndIntegerBit** in Fig. 12.66) the pixel values at the calculated corresponding integer source image coordinates are read from source image. The fractional bit part is used for bilinear interpolation (see **Interpolation**) according to [Bur06] in order to correct the pixel values in the output image due to interpixel positions in the source image. Via DMA the signal is transferred to PC. For demonstration purpose you can see in Fig. 12.86, 12.87 and 12.88 the distorted source image, the Keystone and distortion corrected target image and a rotated and distortion corrected target image.

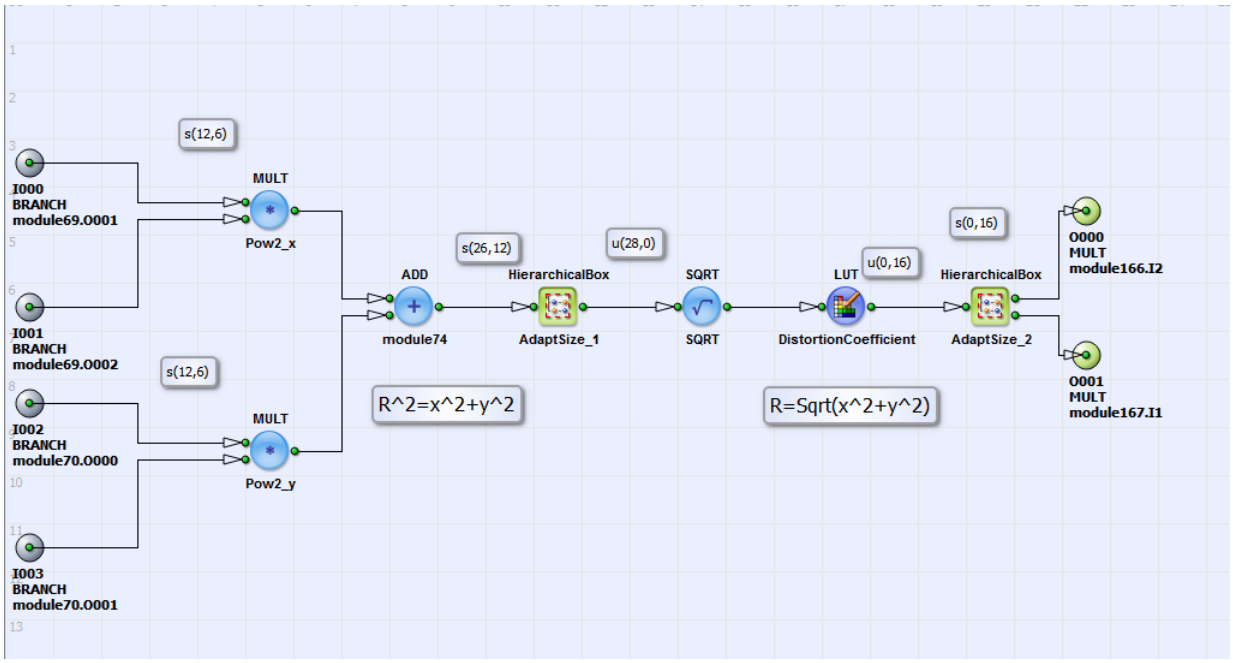



Figure 12.85. Content of HierarchicalBox **DistortionCoefficient**

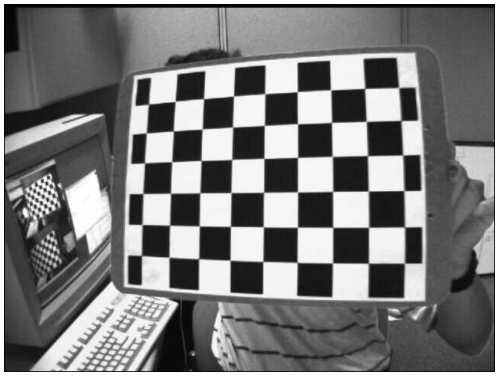


Figure 12.86. Example Source Image [Ope16a]

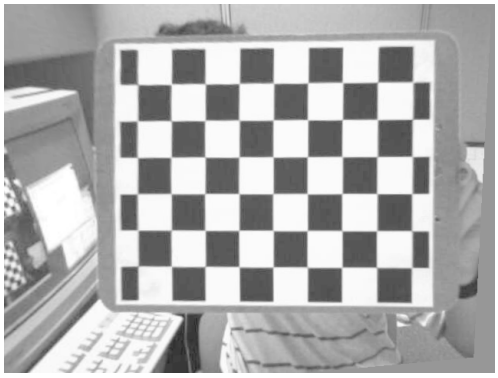


Figure 12.87. Distortion and Keystone corrected target image

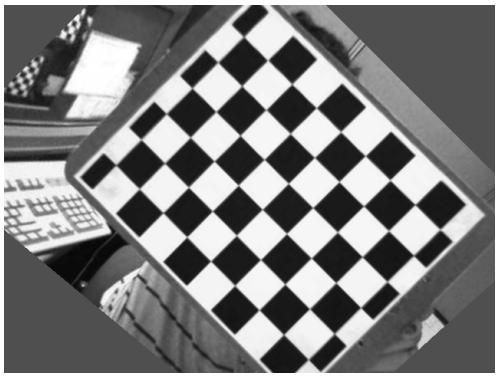
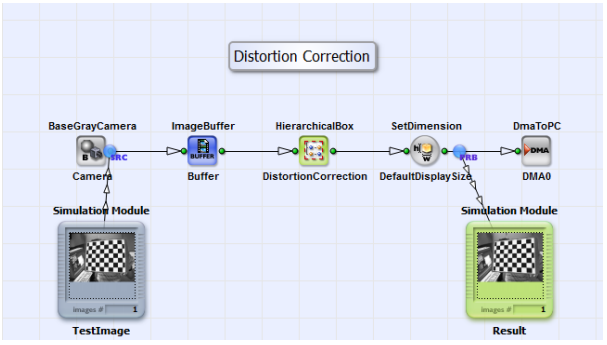


Figure 12.88. Rotated, distortion and Keystone corrected target image

### 12.9.3.2.5. Distortion Correction

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing \Geometry\GeometricTransformation \DistortionCorrection.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>In this example design a distortion correction is implemented. | |

In the design "DistortionCorrection.va" a distortion correction according to eq. 12.22 is implemented. It is analog to the one in the design "GeometricTransformation_DistortionCorrection.va" but without performing geometric transformation and keystone correction. You can see the basic design structure in Fig. 12.89.

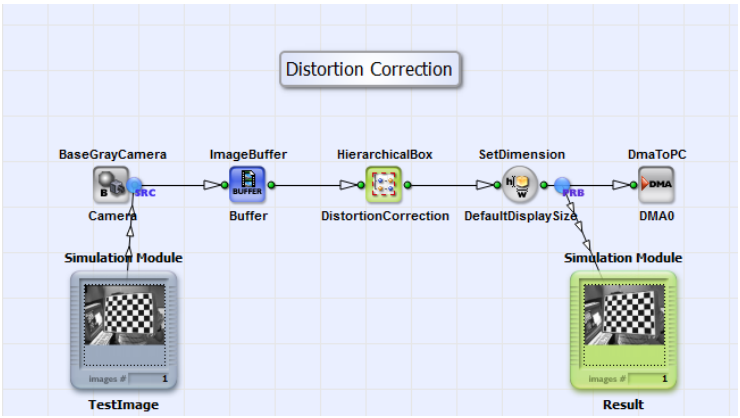



Figure 12.89. Basic design structure of the VA design "DistortionCorrection.va"

The design consists of an interface for a grayscale camera in Camera Link base configuration, a buffer module, the HierachicalBox **DistortionCorrection** and the **DmaToPC**. The distortion correction of the grayscale image is performed in **DistortionCorrection**. You can see its content in Fig. 12.90.

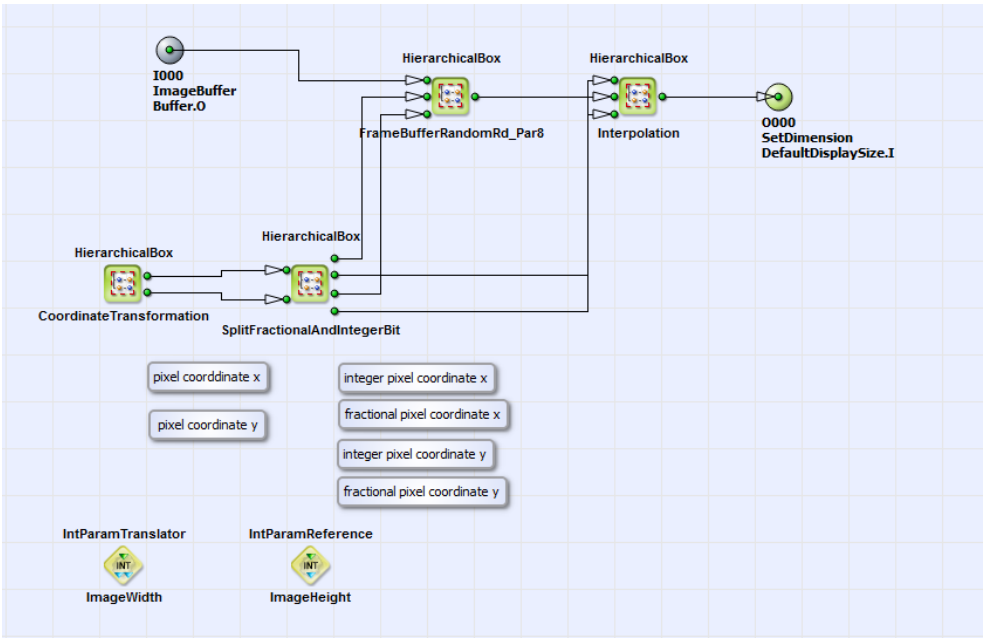



Figure 12.90. Content of HierarchicalBox **DistortionCorrection**

The structure is analog to the content of the HierarchicalBox **GeometricTransformation** (see Fig. 12.66) in the design "GeometricTransformation_PixelReplicator.va" of section 12.9.3.2.1. The operator **FrameBufferRandomRead** is replaced by the HierarchicalBox **FrameBufferRandomRd_Par8** as also done in "GeometricTransformation_PixelReplicator.va" and "GeometricTransformation_DistorationCorrection.va" (see sections 12.9.3.2.3 and 12.9.3.2.4). The source image is linearly written to the buffering element **FrameBufferRandomRd_Par8**. In this module also the three next neighbors are stored together with each pixel (see box **CollectNeighbors** in box **FrameBufferRandomRd_Par8**). In box **CoordinateTransformation**/ **OutputImage** the target image coordinates $x'$, $y'$ are created using the operators **CreateBlankImage**, **CoordinateX** and **CoordinateY**. These coordinates are transformed according to eq. 12.22 to the source image coordinates $x$, $y$ in the HierarchicalBox **InverseCorrection** (see box **CoordinateTransformation**). You can see its content in Fig. 12.91.

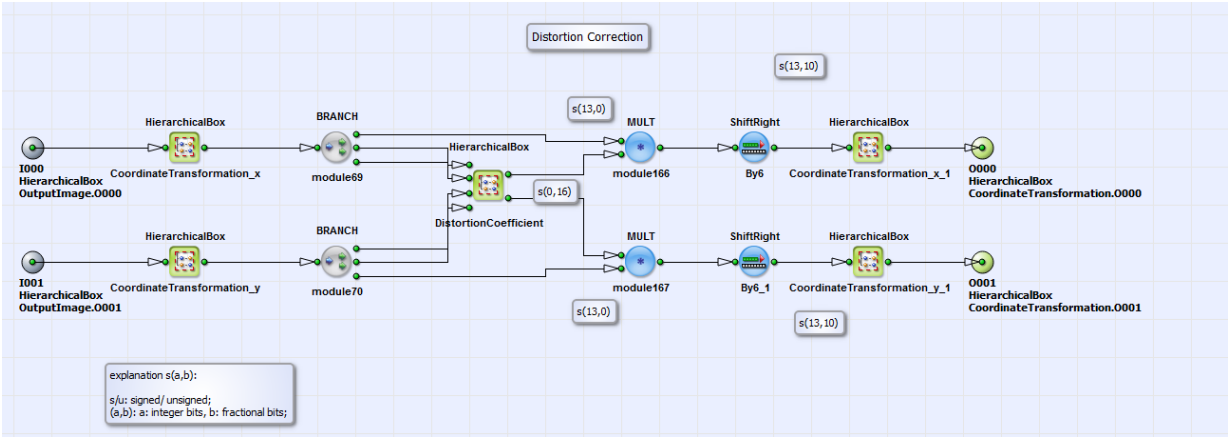
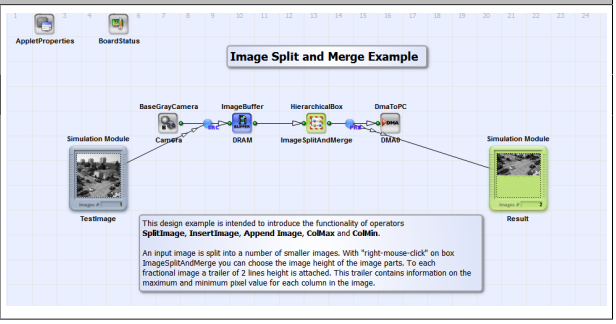



Figure 12.91. Content of HierarchicalBox **InverseCorrection**

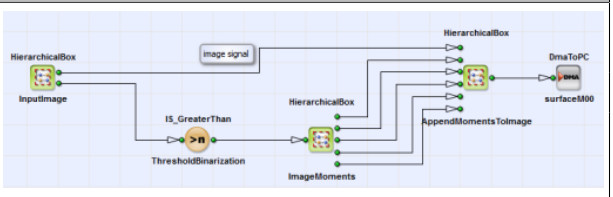
It is equivalent to the implementation of box **DistortionCorrection** of "GeometricTransformation_DistortionCorrection.va" (see Fig. 12.84). For each target image radius $r' = \sqrt{x'^2 + y'^2}$ relative to the image center a lookup table value **C_r** is selected in the HierarchicalBox **DistortionCoefficient**. **C_r** is multiplied with the target image coordinates $x'$, $y'$ according to eq. 12.22 and transformed back to a coordinate system relative to first pixel (see **CoordinateTransformation_x_1** and **CoordinateTransformation_y_1**). As you can see in Fig. 12.90 the resulting source coordinates are split in an integer and fractional part in the box **SplitFractionalAndIntegerBit**. The integer part is used to read the pixels of the source image at the corresponding source image coordinates and the fractional part is used for bilinear interpolation of the final result. The distortion corrected image is the transferred via DMA to PC.

## 12.9.4. ImageSplitAndMerge

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Geometry \ImageSplitAndMerge\ImageSplitAndMerge.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows how to split an merge image streams. Appends a trailer to the image. | |



## 12.9.5. Moments in Image Processing

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Geometry \ImageMoments\ImageMoments.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Calculates the image moments orientation and eccentricity for the incoming images. | |



Moments in image processing are average values from the single pixels` intensities of an image. With this moments physical properties like orientation, eccentricity, the area or the centroid of an object in the image can be identified.

### 12.9.5.1. Orientation $\Theta$

The orientation $\Theta$ of an object is defined as [Bur06]:

$$\Theta = \frac{1}{2} \cdot \tan^{-1}\left(\frac{2\mu_{11}}{\mu_{20} - \mu_{02}}\right), \tag{12.24}$$

with the central moments of second order:

$$\begin{aligned} \mu_{20} &= \frac{M_{20}}{M_{00}} - \bar{x}^2, \\ \mu_{02} &= \frac{M_{02}}{M_{00}} - \bar{y}^2, \\ \mu_{11} &= \frac{M_{11}}{M_{00}} - \bar{x}\bar{y}, \end{aligned} \tag{12.25}$$

with definition

$$\mu_{ij} = \sum_x \sum_y (x - \bar{x})^i \cdot (y - \bar{y})^j \cdot g(x,y) \ . \tag{12.26}$$

The centroid is

$$(\bar{x}, \bar{y}) = (\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}}) \tag{12.27}$$

and the raw moments

$$M_{ij} = \sum_x \sum_y x^i \cdot y^j \cdot g(x,y) \ . \tag{12.28}$$

$M_{00}$ corresponds to the area of an object. Here g(x,y) is the greyscale function for digital greyscale images.

### 12.9.5.2. Eccentricity

The eccentricity e of an object can be calculated with [Bur06]

$$e = \frac{[\mu_{20} \ - \mu_{02}]^2 + 4 \cdot \mu_{11}^2}{[\mu_{20} + \mu_{02}]^2} \tag{12.29}$$

The results of the eccentricity are in a range between 0 (round object) and 1 (elongated object).

### 12.9.5.3. Design in VisualApplets

In VisualApplets a design is implemented with the original image and geometric properties like the area, the center of gravity, the orientation and the eccentricity of an object as output. You can find the example under \examples\Processing\Geometry\ImageMoments\ImageMoments.va. A parallelism of 4 was chosen for calculation. In Figure 12.92, 'Basic design structure' the structure of the design with comments is shown.

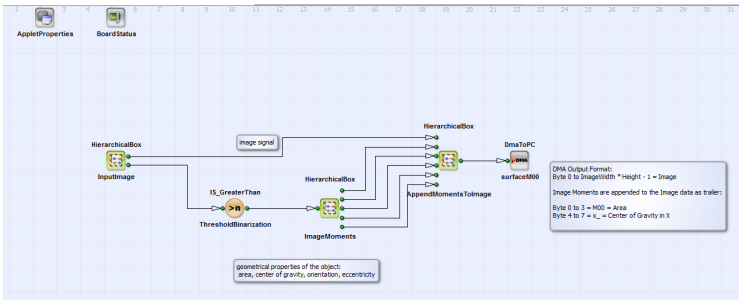



Figure 12.92. Basic design structure

Geometric properties area, center of gravity, orientation and eccentricity of a greyscale image are calculated via moments (HierarchicalBox: **ImageMoments**). These properties are appended to the original camera image (**AppendMomentsToImage**). In Figure 12.93, 'Content of **ImageMoments**' the content of the HierarchicalBox **ImageMoments** is shown. The raw moments $M_{00}$ (which corresponds to the area of an object), $M_{11},\ M_{20},\ M_{02},\ M_{10}$ and $M_{01}$, as well as the center of gravity $(\bar{x}, \bar{y})$ and the central moments $\mu_{20},\ \mu_{02}$ and $\mu_{11}$ are implemented according to the formulas given above. Finally the geometric properties orientation $\Theta$ of the main axis and eccentricity e of an image object are calculated from these results.

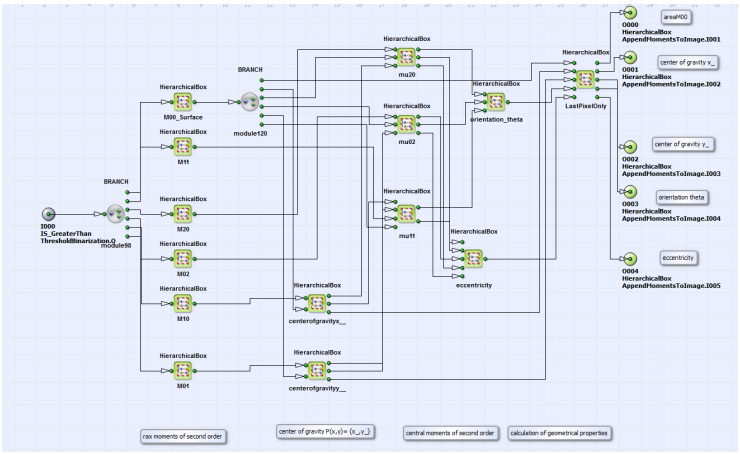



Figure 12.93. Content of **ImageMoments**

In Figure 12.94, 'Content of the HierarchicalBox **orientation_theta**' and Figure 12.95, 'Content of the HierarchicalBox **eccentricity**' the contents of the HierarchicalBoxes **orientation_theta** and **eccentricity** together with inserted comments on the single operation steps are shown. The orientation $\Theta$ and eccentricity e are calculated by using the values of $\mu_{11},\ \mu_{20},\ \mu_{02}$ according to Equation 12.24 and Equation 12.29. A resolution of R=7 is chosen for the input signal to the ARCTAN operator.

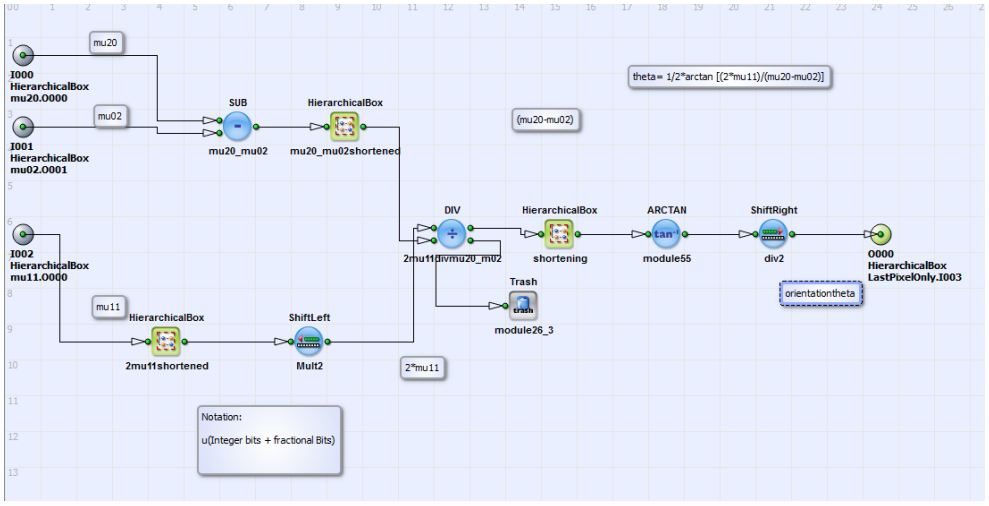



Figure 12.94. Content of the HierarchicalBox **orientation_theta**

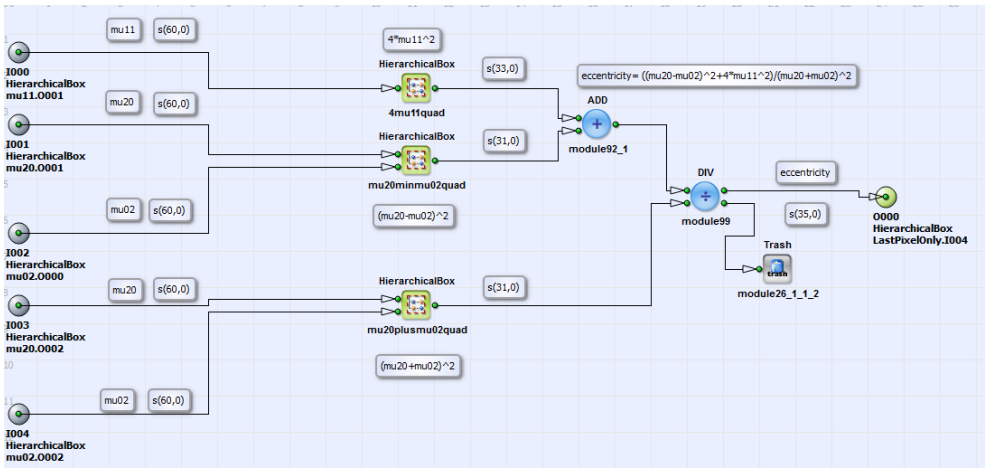
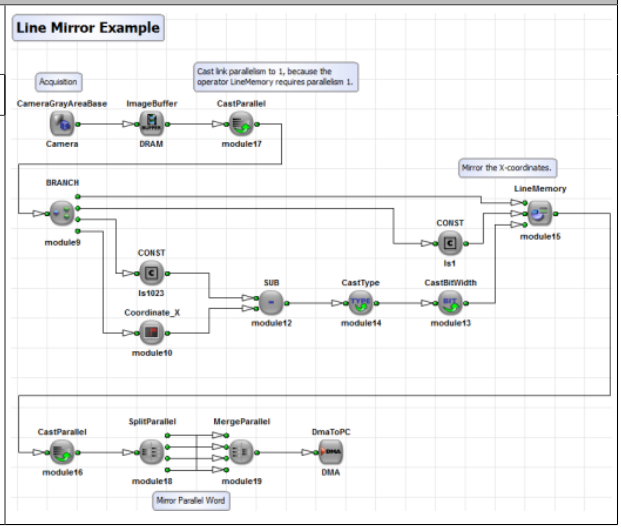


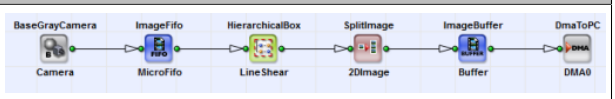

Figure 12.95. Content of the HierarchicalBox **eccentricity**

Comment: During implementation of the orientation of Θ it showed, that limitation to 12 bits maximum input signal of the ARCTAN function can be a problem for calculation of accuracy and resolution: for example the image of an ellipse with orientation Θ= 44.07° (result of moment implementation with MATLAB) was analyzed. The 25 bit input signal for the ARCTAN function has to be limited to 12 bits. With resolution of R= 8 bits the implementation in VisualApplets has result 41.43°; with R=7 bits the result is 43.21° (with resolution for small angles of 0.2°) and with R= 6 it is 44.07° (with resolution for small angles of 0.45°). As an result we found that with high resolution (R= 8 bit) the maximum angle of calculation is smaller than the true angle of orientation, due to bit depth limitation. With smaller resolution R the result of calculation improves for big angles but has as an consequence smaller accuracy in the range of small angles. The problem can be solved by using a Lookup table instead of ARC TAN operator.

## 12.9.6. Line Mirror

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Geometry\LineMirror\LineMirror.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel. | |

## 12.9.7. Shear of an Image

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Geometry\ShearLine\LineShear_V02.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Line shear example with linear interpolation. | |

In this example (under \examples\Processing\Geometry\ShearLine\LineShear_V02.va) a line shear for a line scan camera is implemented. If the camera is not mounted along the intended scanning direction of an object, this design can compensate the resulting shift (see Fig. 12.96) in the scanned object. A line shift of 10 pixel in y direction over the complete image width (here 1024 pixel) is chosen here for example.



Figure 12.96. Skew of a scanned object resulting from camera misalignment

In Fig. 12.97 you can see the basic design structure: Every pixel of the 1D image from the camera operator is sheared in y direction in dependence on its x coordinate in the HierarchicalBox **LineShear**. The single lines are assembled to a 2D image with the operator **SplitImage:2DImage**. The corrected image is sent to PC via the operator **DmaToPC** .

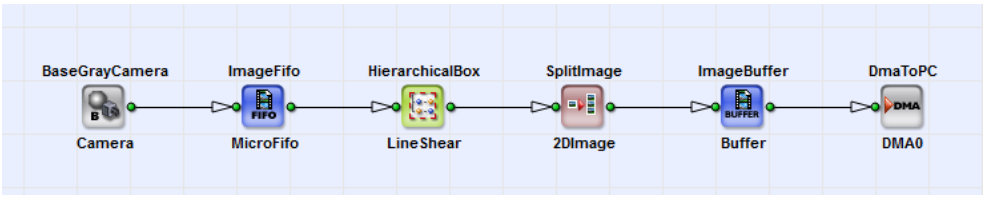



Figure 12.97. Basic design structure

In Fig. 12.98 you can see the content of HierarchicalBox **LineShear**. Here for every pixel the precedent N-1 line neighbors (here 31) are written in a Nx1 kernel.

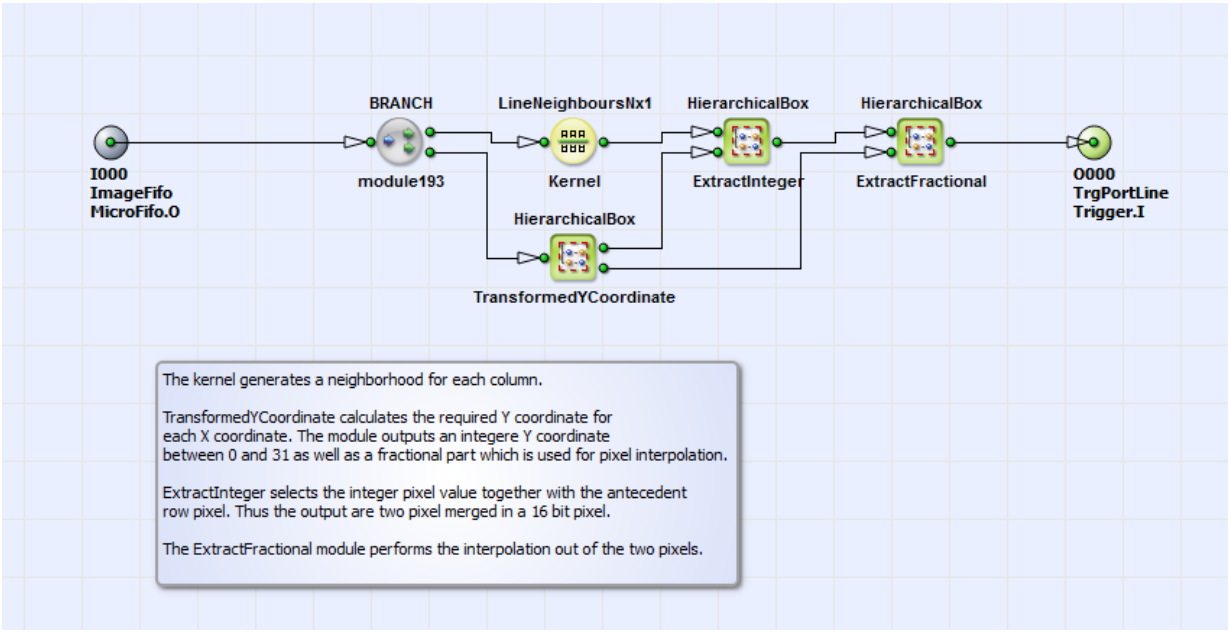



Figure 12.98. Content of HierarchicalBox **LineShear**

Each 8 bit pixel of this column is merged together with its preceding next neighbor (found with the operator **KernelRemap:ShiftedKernel**) into one 16 bit pixel. See therefore the content of HierarchicalBox **ExtractInteger** in Fig. 12.99. In the HierarchicalBox **Select** (see Fig. 12.100) the new pixel value for each pixel is selected with the operator **CASE** in dependence on the shifted (corrected) integer y position.

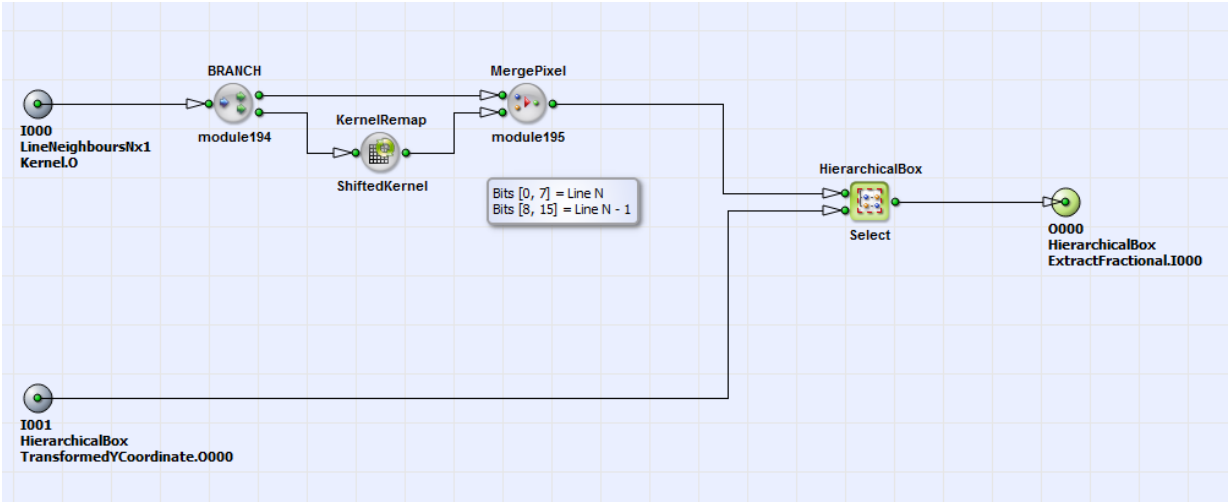



Figure 12.99. Content of HierarchicalBox **ExtractInteger**

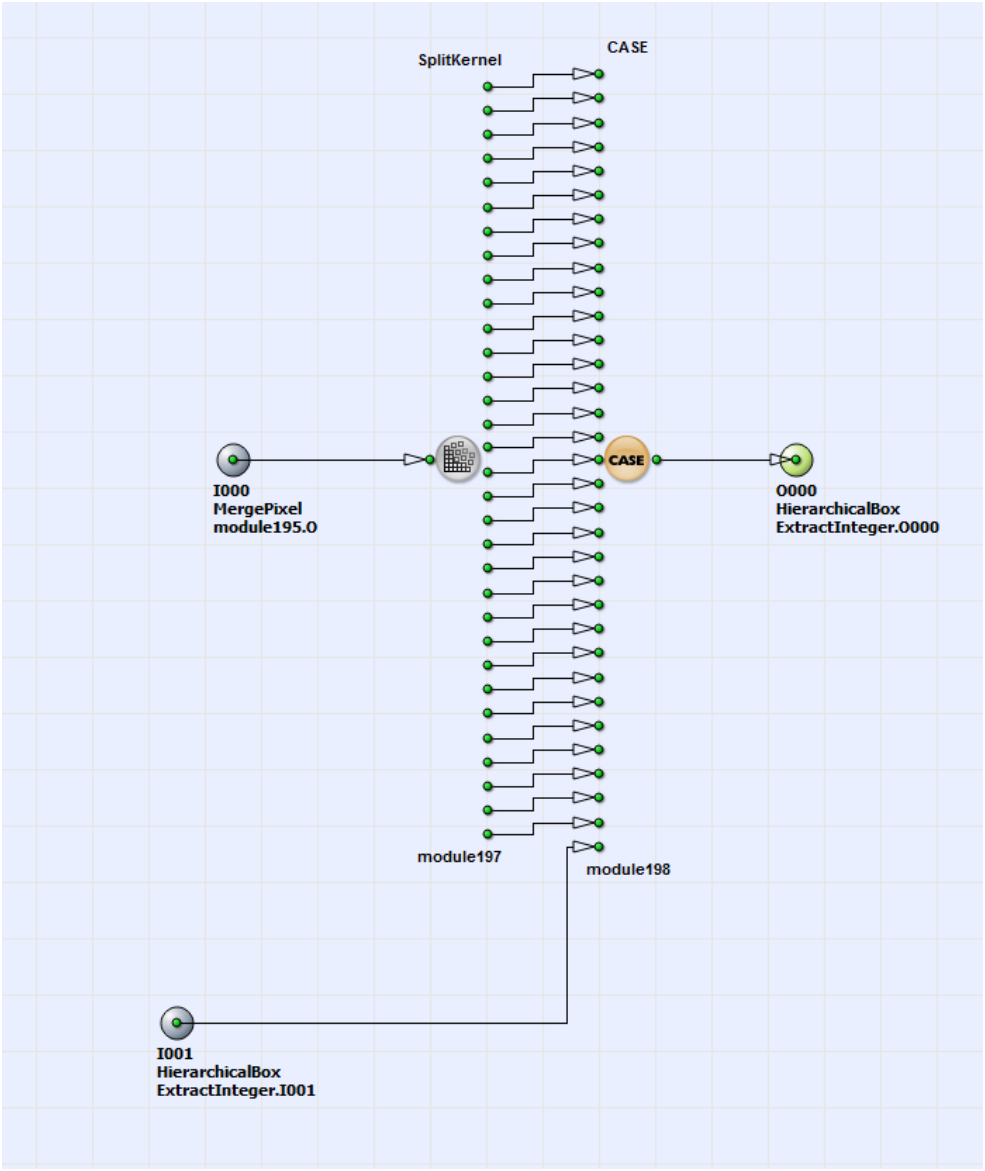



Figure 12.100. Content of HierarchicalBox **Select**

The corrected y position for each pixel is calculated in the HierarchicalBox **TransformedYCoordinate** (see Fig. 12.97). The content of this box is displayed in Fig. 12.101.

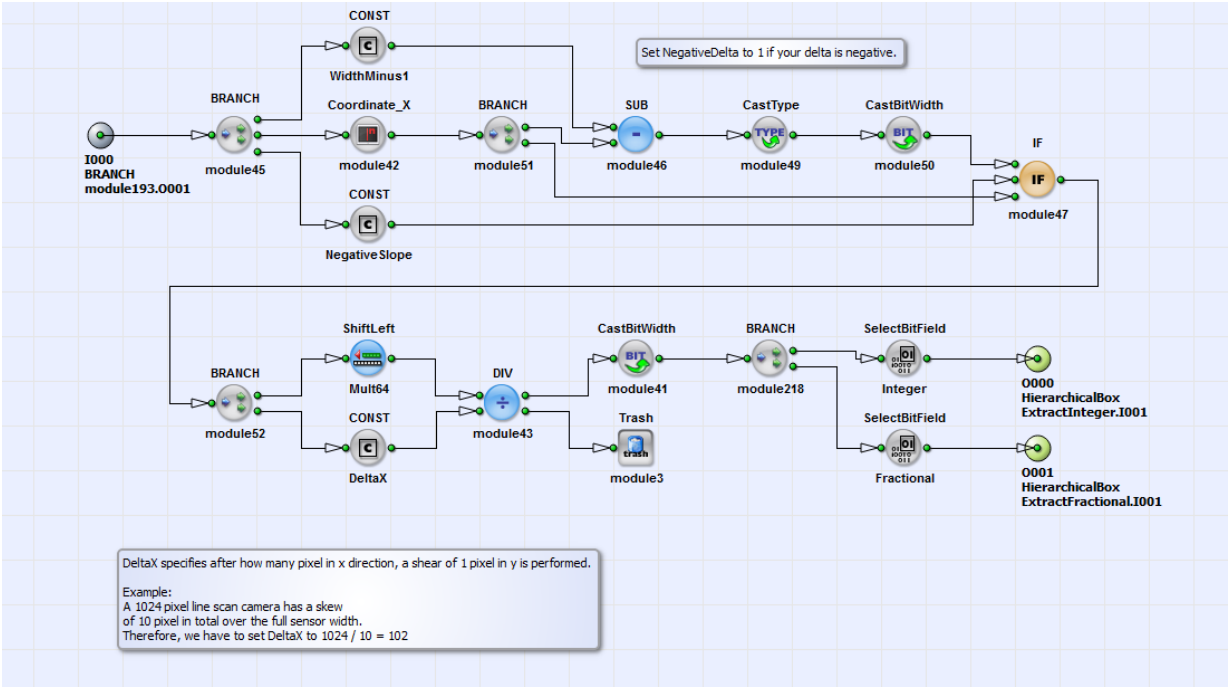


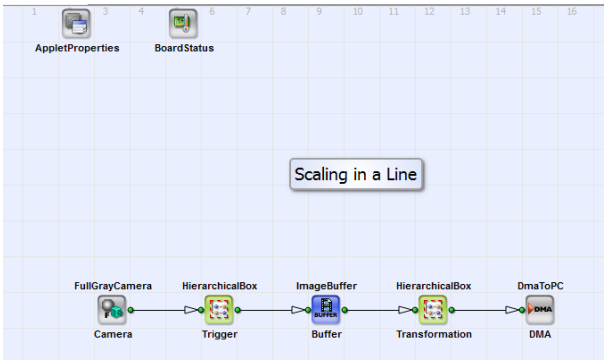

Figure 12.101. Content of HierarchicalBox **TransformedYCoordinate**

With the operator **Coordinate_X** the x position for each pixel is obtained.The possibility to "invert" the x position for negative skew slopes exist (selected with the operators **Const: NegativeSlope** and **IF**). With the operator **CONST: DeltaX** the misalignment of the camera is defined: This constant determines after how many pixels a one pixel shift in y direction is performed. With the operators **SelectBitField: Integer** and **SelectBitField: Fractional** the integer and fractional parts of the corrected y coordinate for every pixel are separated. In this example 6 fractional bits (see operator **ShiftLeft:Mult64**) are determined. The fractional part of each y coordinate is necessary for the linear interpolation performed in the HierachicalBox **Interpolation**, contained in the box **ExtractFractional** (see Fig. 12.98). You can find closer information on the interpolation algorithm in the comment in the HierarchicalBox **Interpolation** in the example. In Fig. 12.102 you can see as a result of the described operations the corrected image.



Figure 12.102. Shift corrected image

## 12.9.8. Scaling a Line Scan Image

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Geometry\ScalingLine\ScalingLineP16.va<br>\examples\Processing\Geometry\ScalingLine\ScalingLineP8.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Scaling and transformation of a line scan image | |

The VisualApplets designs "ScalingLineP8.va" and "ScalingLineP16.va" scale an image of a grayscale line scan camera in CameraLink Full configuration by an arbitrary factor to transform the width of an image between the input and the output. „ScalingLineP8.va" is designed for a parallelism of 8 and "ScalingLineP16.va" for a parallelism of 16 for a marathon VCL platform. This document describes the algorithm and implementation in VisualApplets. For usage of the design, a number of lookup table values need to be calculated. An included C++ and Matlab program provide these calculations to simplify the usage.

### 12.9.8.1. Basic Idea for Scaling/Transformation in a Line

The scaling/transformation/distortion correction of a line image is performed on the basis of source and target image coordinates. In the following the basic principle of the algorithm is explained with example coordinate pairs and parallelism 4.

Target coordinates: 0 1 2 3 | 4 5 6 7 | 8 9 10 11 | 12 13 14 15
Source coordinates: 0.3 0.5 1.4 1.5 | 2.2 3.4 3.6 4.2| 5.4 5.5 7.8 8.5 | 9.1 11.4 13.3 14.1

We use the Target-To-Source procedure. For every pixel in the target image (= output image to PC) we select the corresponding (rounded off) integer coordinate and the corresponding pixel value in the source image. Positions between pixels are considered by selecting also the preceding pixel neighbor of the current pixel coordinate in the source image. With linear interpolation the correct pixel value in the output image is calculated. Due to parallelism single pixels can not be read from source image. Therefore "words" (unities of pixels read at the same time) have to be read in a useful sequence. It may happen that words have to be read more than once or never. Within these words, which have been read from the source image, the correct pixel positions for the creation of the output word have to be selected. Pixels not needed from the words read are deleted. The following example illustrates the read access from source image coordinates:

| Memory Read Count cycles | Requested Target Coordinates | Respective Source Coordinates | Read Word/ incl. Pixel | Use Pixel in Word |
|---|---|---|---|---|
| 1 | 0, 1 2, 3 | 0.3, 0.5, 1.4, 1.5 | 0/ 0, 1, 2, 3 | 1, 1, 2, 2 |
| 2 | 4 | 2.2 | 0/ 0, 1, 2, 3 | 3, x, x, x |
| 3 | 5, 6, 7, 8 | 3.4, 3.6, 4.2, 5.4 | 1/ 4, 5, 6, 7 | 4, 4, 5, 6 |
| 4 | 9 | 5.5 | 1/ 4, 5, 6, 7 | 6, x, x, x |
| 5 | 10, 11, 12 | 7.8, 8.5, 9.1 | 2/ 8, 9, 10, 11 | 9, 8, 10, x |
| 6 | 13, 14, 15 | 11.4, 13.3, 14.1 | 3 / 12, 13, 14, 15 | 12, 14, 15, x |

Table 12.6. Reading Cycles

While "x" indicates that the pixel is deleted.

### 12.9.8.2. Implementation in VisualApplets

The main components of the design are (Fig. 12.103):

1. Interface to line scan camera including trigger system and buffer;
2. Scaling of camera image (in **Transformation**);
3. DMA data transfer to PC;

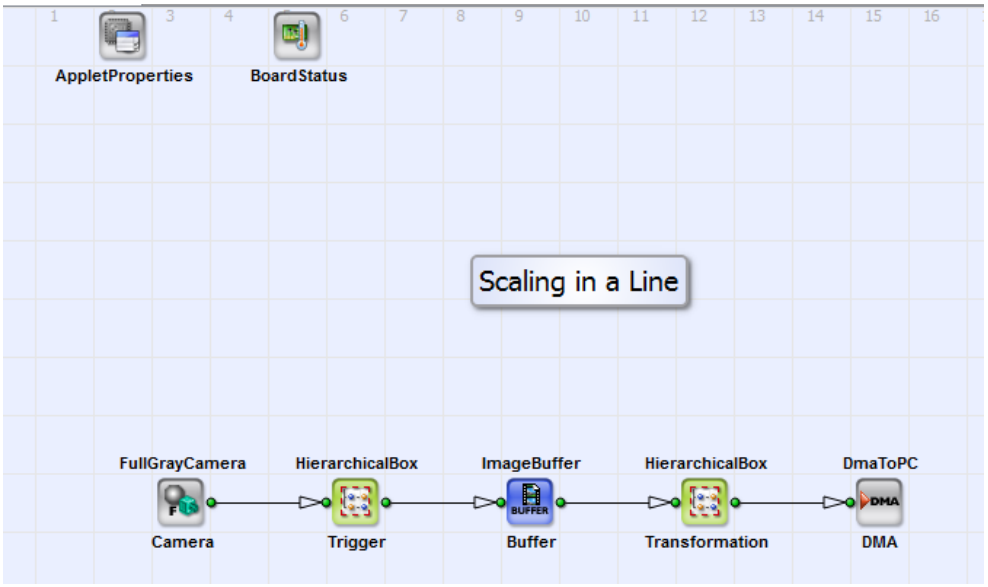



Figure 12.103. Basic design structure for scaling a line camera image

In the following we will describe the main component **Transformation** in detail.

### 12.9.8.2.1. Transformation

You can see the components of **Transformation** in Fig. 12.104.

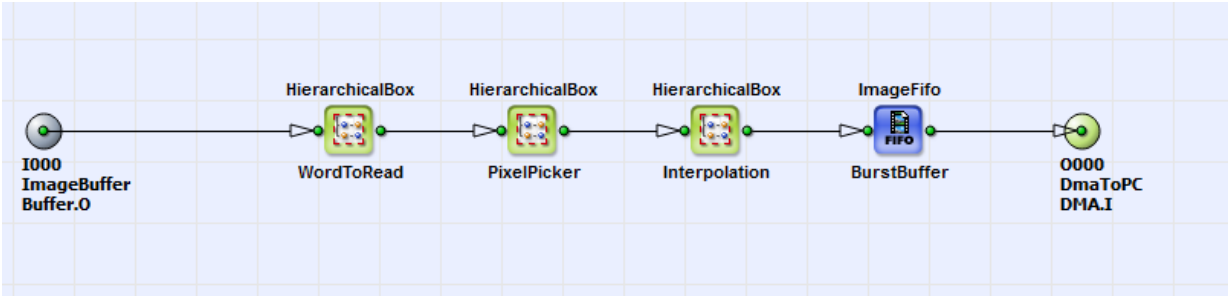



Figure 12.104. Components of **Transformation**

The scaling of a camera line scan image is operated in Target-To-Source procedure. In WordToRead we select in which useful order the unities of 8 parallelism 8 in „ScalingLineP8.va“) or 16 (parallelism 16 in „ScalingLineP16.va“) pixels (= ”word”) are read from input camera image. In **PixelPicker** we decide which pixels we need from the words read. **Interpolation** considers for the calculation of the pixel values in the output image the interpixel position in the source image.

#### 12.9.8.2.1.1. WordToRead

The components of the module WordToRead are shown in Fig. 12.105.

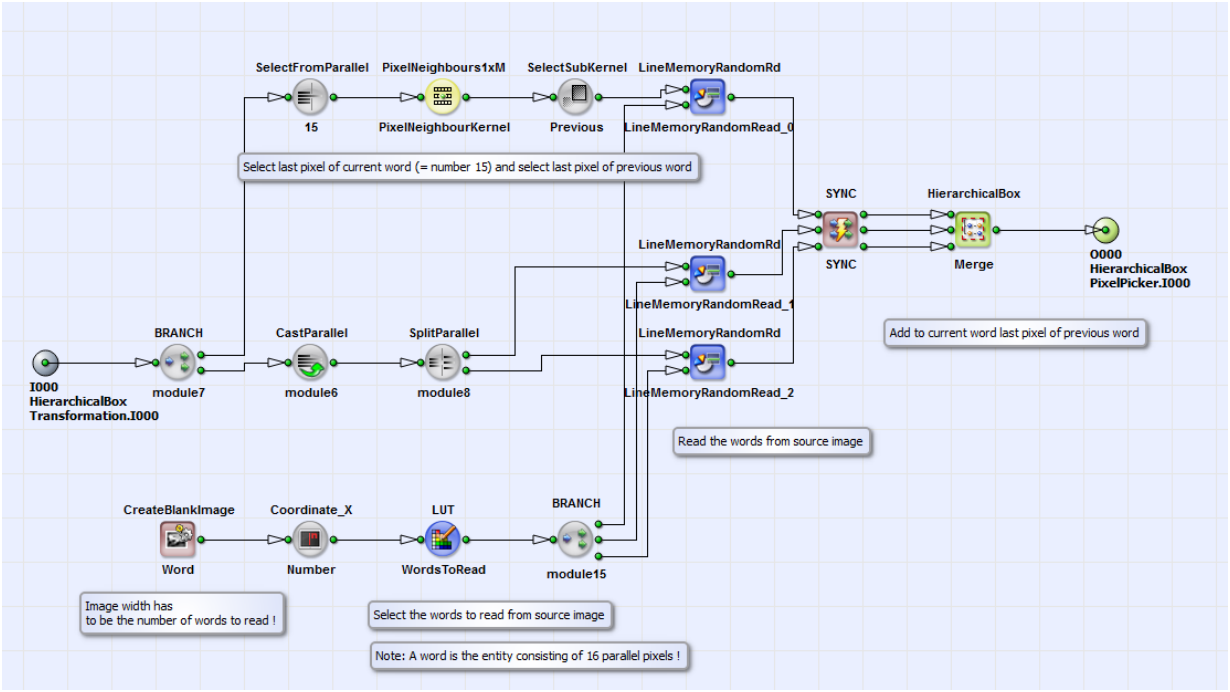



Figure 12.105. Components of **WordToRead**

The order and frequency of the words to be read from the source image is defined in **LUT: WordsToRead**. The content of this table is created with the external C++ program „ScalingLUTS.cpp" or alternative with the MATLAB modules "LUTS_Scaling.m" and "ScalingTablelLine.m". See therefor section 12.9.8.3 in this document. The input link for this lookup table is defined with **CreateBlankImage** and **Coordinate_X**. The value of parameter **ImageWidth** of **CreateBlankImage** has to be exactly the same as the number of elements in **LUT: WordsToRead**. The parameter **ImageHeight** has to match the image dimensions of the source image in y-direction. The words from the source image are read from buffers **LineMemoryRandomRd_1** and **LineMemoryRandomRd_2** in an order defined by **LUT: WordsToRead**. The source image is linearly written to these buffers. Here two buffers are necessary due to a maximum bit depth of 64 bit and parallelism 1. For every word the last pixel of the preceding word is selected and read (upper operation line before **SYNC**). This value is added to the corresponding word with the operator **Merge**.

### 12.9.8.2.1.2. PixelPicker

In the module **PixelPicker** (see Fig. 12.106) the pixels for the scaled output words are selected with the components **Pick_0** to **Pick_15** (or to **Pick_7** with parallelism 8) from the words read. If pixels of a word read are not needed (see section 12.9.8.1) the lookup table **RemovePixel** deletes them. The content of the tables named above (**Pick_0** to **Pick_15** and **RemovePixel**) can be created with the external C++ program „ScalingLUTS.cpp" or alternative with the MATLAB modules "LUTS_Scaling.m" and "ScalingTablelLine.m". See therefor also section 12.9.8.3 in this document.

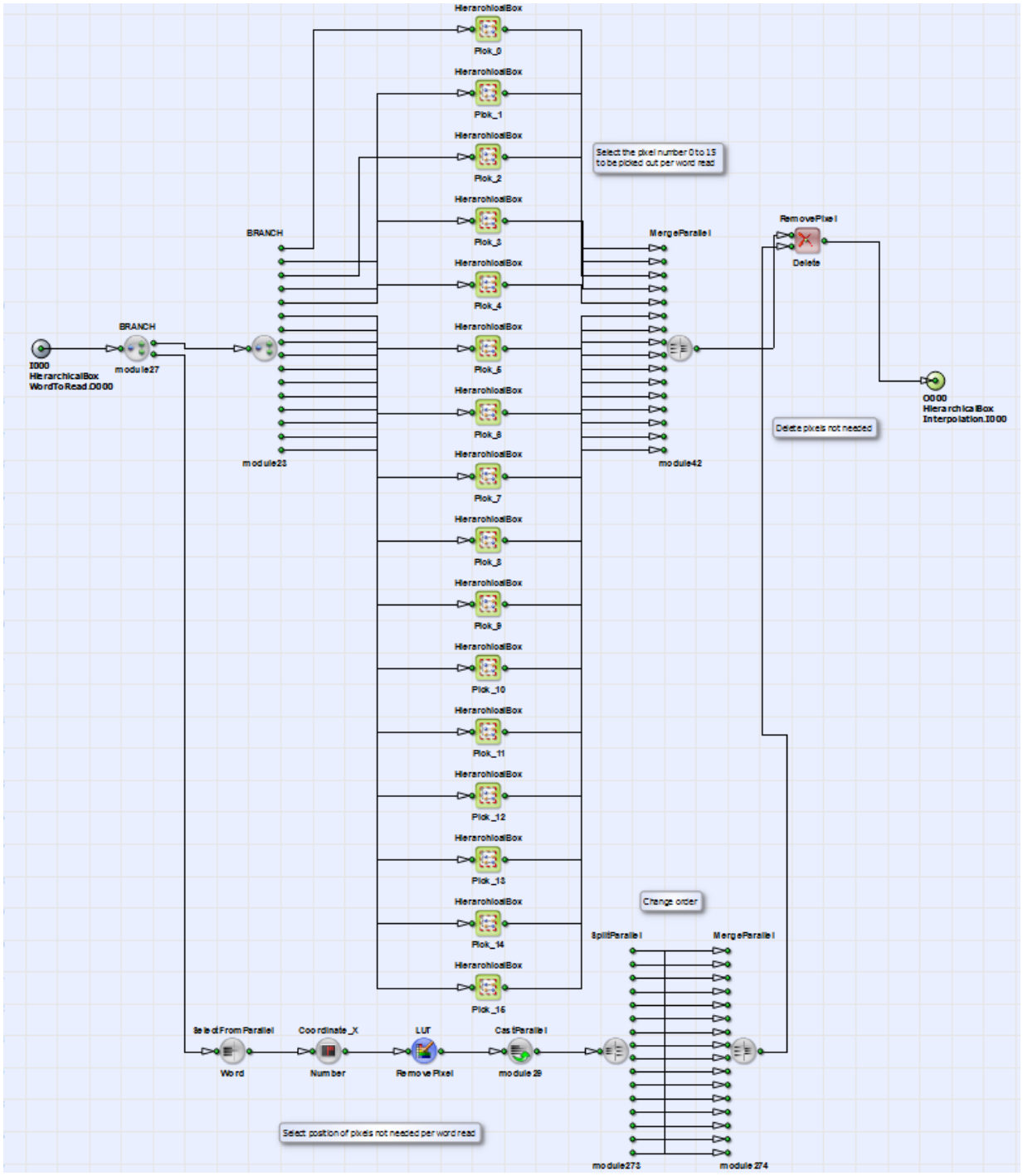



Figure 12.106. Components of **PixelPicker**

In Fig. 12.107 the content of the HierarchicalBox **Pick_0** is shown. The lookup table **PickComponent0** defines which pixel of a word read (**SelectFromParallel: Word** and **Coordinate_X: Number**) is the 0. component of the current output word. With the operator **CASE** the corresponding pixel is selected. Together with its predecessor it is written to the current pixel (**MergePixel**). Analog to this procedure described, the pixels 1 to 15 (or 7 with parallelism 8) of the current output word are selected with the modules **Pick_1** to **Pick_15** (or to **Pick_7**). Die single pixels **Pick_0** to **Pick_15** (or to **Pick_7**) are combined to the current transformed/scaled word with parallelism 16 or 8 (**MergeParallel** in Fig. 12.106).

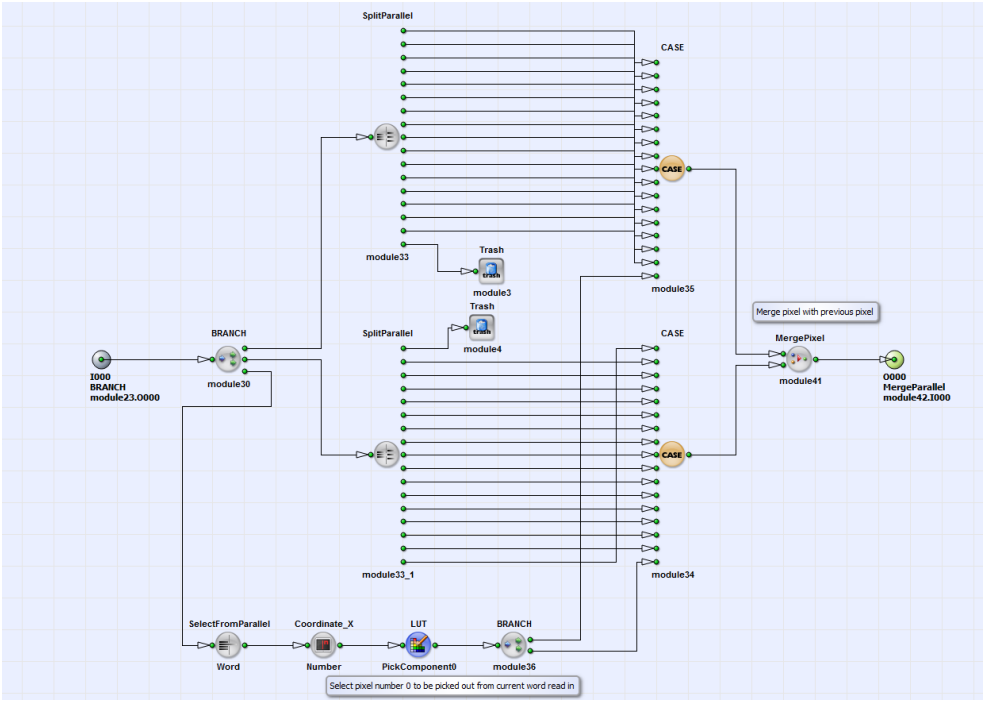



Figure 12.107. Content of **Pick_0**

## 12.9.8.2.1.3. Interpolation

In order to obtain the correct value on the pixel positions in the output image sent to PC, the interpixel position in the source image has to be considered. In the module **Interpolation** (Fig. 12.104) a linear interpolation is performed [Bur06]:

$$f_c = f(x) = (1 - dx) \cdot f(x_1) + dx \cdot f(x_2) \ . \tag{12.30}$$

Here $f_c$ is the pixel value in the output image, which corresponds to value $f(x)$ in the source image at position x. Furthermore $f(x_1)$ and $f(x_2)$ are the values at pixel coordinates $x_1$ and $x_2$ in the source image. The interpixel position is $dx$. More information on linear interpolation you can find e.g. in [Bur06].

The content of the module **Interpolation** is shown in Fig. 12.108. The values of current pixels $f(x_1)$ and their predecessors $f(x_2)$ are separated with the operators **SelectBitField: Current** and **SelectBitField: Previous**. The values of $f(x_1)$ and $f(x_2)$ are multiplied with $(1 - dx)$ and $dx$ according to eq. 12.30. The information for the interpixel position $dx$ is contained in four lookup tables **Fractional_Part0** to **Fractional_Part3** in the design "ScalingLineP16.va". Every table contains for 4 pixels of the current output word the interpixel position $dx$. Every interpixel position has 6 bit depth. The content of the LUTs **Fractional_Part0** to **Fractional_Part3** can be created with the external C++ program „ScalingLUTS.cpp" or alternative with the MATLAB modules "LUTS_Scaling.m" and "ScalingTablelLine.m". See therefor also section 12.9.8.3 in this document. The design "ScalingLineP8.va" has one table Fractional containing the interpixel position. Its content can also be created with the external C++ and Matlab program modules. Finally the transformed/scaled line image is sent via DMA transfer to PC (see Fig. 12.104).

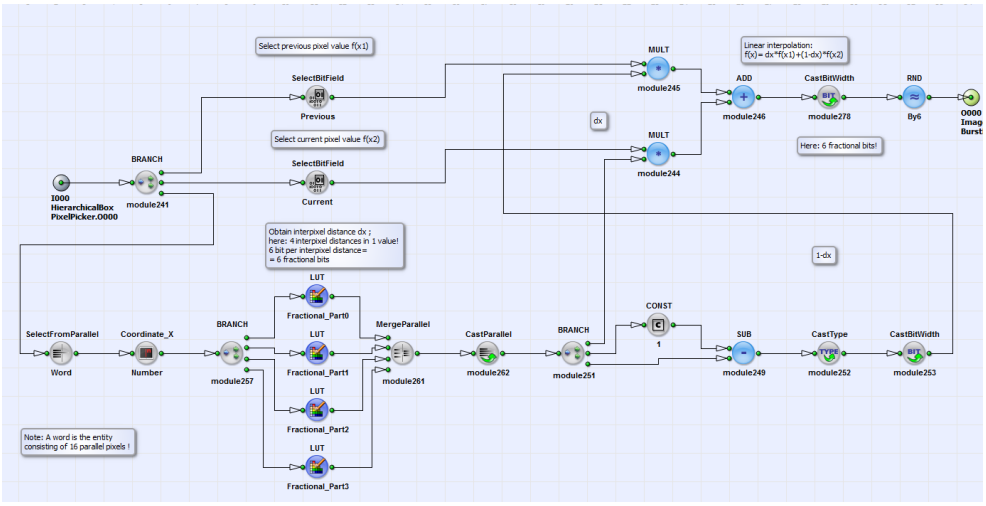



Figure 12.108. Components of **Interpolation**

## 12.9.8.3. Lookup Tables for The VisualApplets Design

The lookup tables necessary for the design "ScalingLineP16.va" and "ScalingLineP8.va" (see section 12.9.8.2) can be created either with the C++ program "ScalingLUTs.cpp" (in folder "LUTS_Cpp") or with the MATLAB program modules "LUTS_Scaling.m" and "ScalingTableLine.m"(in folder "LUTS_MATLAB").

### 12.9.8.3.1. ScalingLUTs.cpp

1. Please define following parameters in the program (or in the command prompt):

//////////////////////Parameter//////////////////////

/ Please specify these parameters!//////////

uint64_t parallelism = 8; // =8 for "ScalingLineP8.va" and =16 for "ScalingLineP16.va"

uint64_t fractionalBits = 6; // = 6: default value in "ScalingLineP8.va" and "ScalingLineP16.va"

uint64_t noOfFractionalLUTs = parallelism == 8 ? 1 : 4; // number of fractional tables = 1 in "ScalingLineP8.va" and = 4 in "ScalingLineP16.va"

int InputWidth; // width of input image

int OutputWidth; // width of image sent to PC

For the VisualApplets design "ScalingLineP16.va" set the parallelism parameter to 16 and for"ScalingLineP8.va" to 8. The number of fractionalBits of 6 is a default value in both designs. The design "ScalingLineP16.va"has four tables (const int noOfFractionalLUTs = 4) containing the interpixel positions for the interpolation (see Fig. 12.108). The design "ScalingLineP8.va" has one table (const int noOfFractionalLUTs = 1) instead. The scaling of your line image is defined by the relation between InputWidth and OutputWidth.

2. Run the program!

3. Following files are created which can be loaded to the lookup tables in the VisualApplets designs:

| File | VA LUT |
|---|---|
| **FractionalTable_0.txt** to **FractionalTable_3.txt** | **Fractional_Part0** (or Fractional for parallelism 8) to **FractionalPart_3** |

| File | VA LUT |
|---|---|
| **PickerTable_0.txt** to **PickerTable_15.txt** (or to **PickerTable_7.txt**) | **PickComponent0** to **PickComponent15** (or to **PickComponent7**) |
| **TableWordsToRead.txt** | **WordsToRead** |
| **RemoveTable.txt** | **RemovePixel** |

Table 12.7. Files and their corresponding lookup tables in Visual Applets

In addition the text file **ScalingTable.txt** is created. It contains for every output image pixel the corresponding coordinates in the source image (see also section 12.9.8.1).

### 12.9.8.3.2. ScalingTableLine.m and LUTS_Scaling.m

Please open the module "ScalingTableLine.m" and define the parameters

InputWidth=…;

OutputWidth=…;

1. Run the program!

2.The text file ScalingTableLine.txt is created. It contains for every output image pixel the corresponding coordinates in the source image.

3. Please open the module "LUTS_Scaling.m" and define the parameters

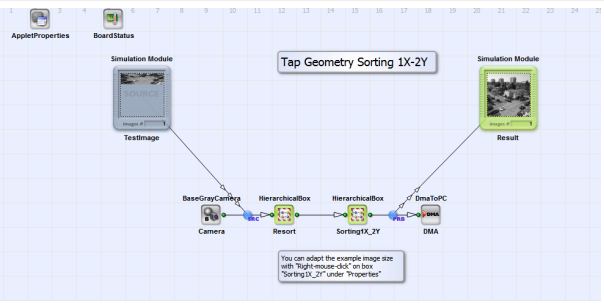
parallelism = ...;

NoOfFractionalBits=...;

NoOfFractionalTables=...;

Run the program! The files created correspond to the files in Tab. 12.7 which can be loaded to the lookup tables in the VisualApplets designs.

## 12.9.9. Tap Geometry Sorting

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Geometry \TapGeometrySorting\TapSorting_2X_1Y.va \examples\Processing\Geometry \TapGeometrySorting\TapSorting_2XE_1Y.va \examples\Processing\Geometry \TapGeometrySorting\TapSorting_1X_2Y.va \examples\Processing\Geometry \TapGeometrySorting\TapSorting_2X_2Y.va \examples\Processing\Geometry \TapGeometrySorting\TapSorting_2X_2YE.va \examples\Processing\Geometry \TapGeometrySorting\TapSorting_8X_1Y.va \examples\Processing\Geometry \TapGeometrySorting\TapSorting_10X_1Y.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Sorting of Camera Link Taps | |

### 12.9.9.1. Small Theory on Camera Link Tap Geometry

Taps are geometric zones on a camera sensor. The pixels of a frame are transmitted pixel by pixel to the frame grabber in sequential order from the taps. The pixels can be transferred in parallel from these taps. In Camera Link base configuration the number of taps transferred at the same clock cycle are 1 to 3, in medium configuration 3 or 4 and in full configuration 8 or 10. The geometric arrangement of the taps depends on the sensor model. Depending on this order, the pixels in the acquired image need to be resorted in order to achieve an image, which mirrors reality. To describe tap geometry configuration of a camera a naming convention

$$< \mathrm{RegionX} > \mathrm{X}(< \mathrm{TapX} >)(< \mathrm{ExtX} >)\text{-} < \mathrm{RegionY} > \mathrm{Y}(< \mathrm{TapY} >)(< \mathrm{ExtY} >) \tag{12.31}$$

is used. Hereby $< RegionX > and < RegionY >$ are the number of taps in horizontal and vertical direction. The number of consecutive pixels in X and Y direction, which are transferred simultaneously from a tap are $< TapX >$ and $< TapY >$. $< ExtX >$ and $< ExtY >$ can be named "E", "M" or "R". "E" indicates, that the readout per taps starts from both ends of pixel lines/columns. "M" means, that the pixel extraction starts from the middle of line. "R" shows, that pixel extraction starts at the right side of each tap. In table 12.8 the pixel positions in horizontal and vertical direction for an image with width "w" and "height" h for specific tap geometries are listed.

| Tap Geometry | X Start | X End | X Step | Y Start | Y End | Y Step | |
|---|---|---|---|---|---|---|---|
| 2X-1Y | Tap 1 | 1 | w/2 | 1 | 1 | h | 1 |
| | Tap 2 | w/2+1 | w | 1 | 1 | h | 1 |
| 2XE-1Y | Tap 1 | 1 | w/2 | 1 | 1 | h | 1 |
| | Tap 2 | w | w/2+1 | 1 | 1 | h | 1 |
| 1X-2Y | Tap 1 | 1 | w | 1 | 1 | h/2 | 1 |
| | Tap 2 | 1 | w | 1 | h/2+1 | h | 1 |
| 2X-2Y | Tap 1 | 1 | w/2 | 1 | 1 | h/2 | 1 |
| | Tap 2 | w/2+1 | w | 1 | 1 | h/2 | 1 |
| | Tap 3 | 1 | w/2 | 1 | h/2+1 | h | 1 |
| | Tap 4 | w/2+1 | w | 1 | h/2+1 | h | 1 |
| 2X-2YE | Tap 1 | 1 | w/2 | 1 | 1 | h/2 | 1 |
| | Tap 2 | w/2+1 | w | 1 | 1 | h/2 | 1 |
| | Tap 3 | 1 | w/2 | 1 | h | h/2+1 | 1 |
| | Tap 4 | w/2+1 | w | 1 | h | h/2+1 | 1 |
| 8X-1Y | Tap 1 | 1 | w/8 | 1 | 1 | h | 1 |
| | Tap 2 | w/8+1 | 1/4 w | 1 | 1 | h | 1 |
| | Tap 3 | 1/4 w +1 | 3/8 w | 1 | 1 | h | 1 |
| | Tap 4 | 3/8 w +1 | 1/2 w | 1 | 1 | h | 1 |
| | Tap 5 | 1/2 w +1 | 5/8 w | 1 | 1 | h | 1 |
| | Tap 6 | 5/8 w +1 | 3/4 w | 1 | 1 | h | 1 |
| | Tap 7 | 3/4 w +1 | 7/8 w | 1 | 1 | h | 1 |
| | Tap 8 | 7/8 w +1 | w | 1 | 1 | h | 1 |
| 10X-1Y | Tap 1 | 1 | w/10 | 1 | 1 | h | 1 |
| | Tap 2 | w/10+1 | 1/5 w | 1 | 1 | h | 1 |

| Tap Geometry | X Start | X End | X Step | Y Start | Y End | Y Step | |
|---|---|---|---|---|---|---|---|
| | Tap 3 | 1/5 w +1 | 3/10 w | 1 | 1 | h | 1 |
| | Tap 4 | 3/10 w +1 | 2/5 w | 1 | 1 | h | 1 |
| | Tap 5 | 2/5 w +1 | 1/2 w | 1 | 1 | h | 1 |
| | Tap 6 | 1/2 w +1 | 3/5 w | 1 | 1 | h | 1 |
| | Tap 7 | 3/5 w +1 | 7/10 w | 1 | 1 | h | 1 |
| | Tap 8 | 7/10 w +1 | 4/5 | 1 | 1 | h | 1 |
| | Tap 9 | 4/5 w +1 | 9/10 | 1 | 1 | h | 1 |
| | Tap 10 | 9/10 w +1 | w | 1 | 1 | h | 1 |

Table 12.8. Examples of tap geometries

### 12.9.9.2. Implementation in VisualApplets

The sorting of the tap geometry modi described above is implemented in seven seperate Visual Applets designs. The designs can be used as part of image processing designs. For the simulation and test of the tap geometry sorting you find example images under \examples\Processing\Geometry \TapGeometrySorting\TestImages. The implementation algorithm differs from design to design to find the best balance between complexity and resource efficiency of the implementation. The simplest way to realize tap geometry sorting in horizontal direction is realized in the designs "TapSorting_2X_1Y.va" and "TapSorting_2XE_1Y.va". You can see the basic design structure of "TapSorting_2XE_1Y.va" in Fig. 12.109.

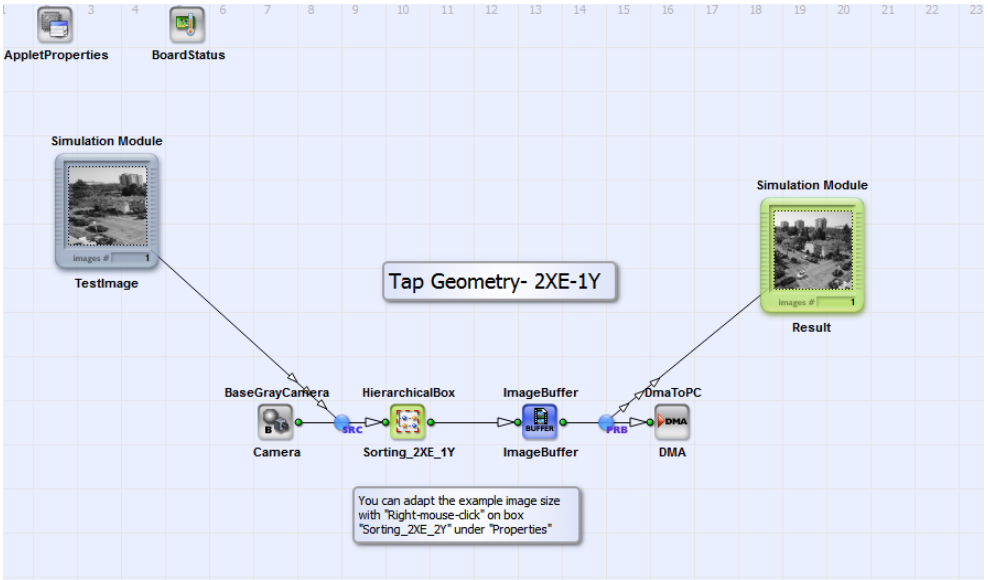



Figure 12.109. Basic design structure for "TapSorting_2XE_1Y.va"

The pixels of the two taps "2X" in horizontal direction coming from a grayscale camera in Camera Link base configuration are split in two branches. The pixels of tap 2 are mirrored in horizontal direction in the HierachicalBox **MirroredLine** using operator **LineMemory**. The pixels of the two

taps are then inserted line by line to a new image. Two sequential lines are then combined to one bigger line. This image is the output image with the correct tap geometry: The pixels of tap 1 are in the left half of the result image, whereas the pixels of tap 2 are in the right half in reverse order due to mirroring. The design structure and algorithm of "TapSorting_2X_1Y.va" is analog to the one of "TapSorting_2XE_1Y.va" but without the mirroring of the pixels of tap 2. The designs "TapSorting_1X_2Y.va", "TapSorting_2X_2Y.va" and "TapSorting_2X_2YE.va" have one or two taps in horizontal ("1X" or "2X") and two taps in vertical direction ("2Y" and "2YE") . The most efficient way of tap sorting is here using operator **FrameBufferRandomRead**. You can see the basic design structure of "TapSorting_2X_2Y.va" in Fig. 12.110. The design structures of "TapSorting_1X_2Y.va" and "TapSorting_2X_2YE.va" are equivalent.

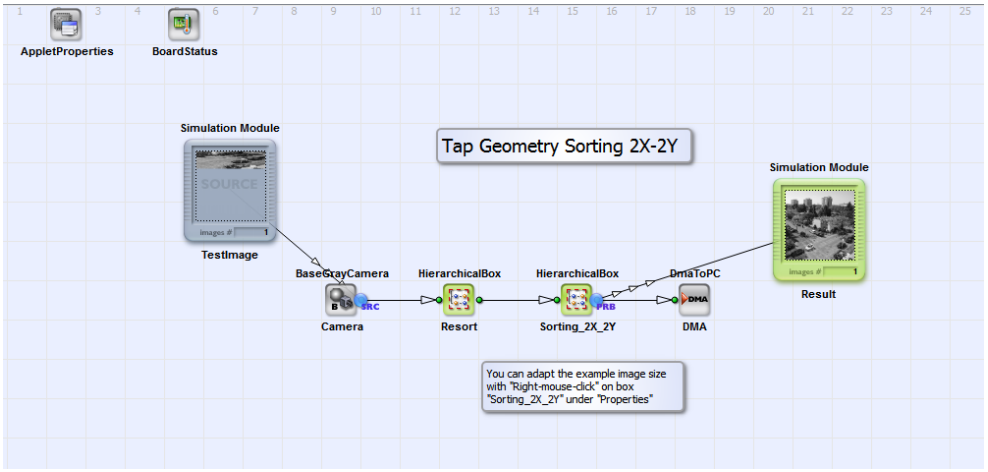



Figure 12.110. Basic design structure of "TapSorting_2X_2Y.va"

The pixels transferred from a camera in camera link base configuration are rearranged in their order in the HierarchicalBox **Resort**. Here four pixels of the same tap are merged to one pixel. Tap 1 to Tap 2 ("TapSorting_1X_2Y.va") or to Tap 4 ("TapSorting_2X_2Y.va" and "TapSorting_2X_2YE.va") are merged in parallel. Via the operator **FrameBufferRandomRead** with subsequent reinterpretation of the pixel depth (operator **CastParallel**) the pixels of the camera taps are positioned correctly in the result image. The correct address input for **FrameBufferRandomRead** is implemented in box **Address**. Here the designs differ in detail in dependence on the tap geometry used but follow the same principle. In Fig. 12.111 you can see the address generation for the design "TapSorting_2X_2Y.va".

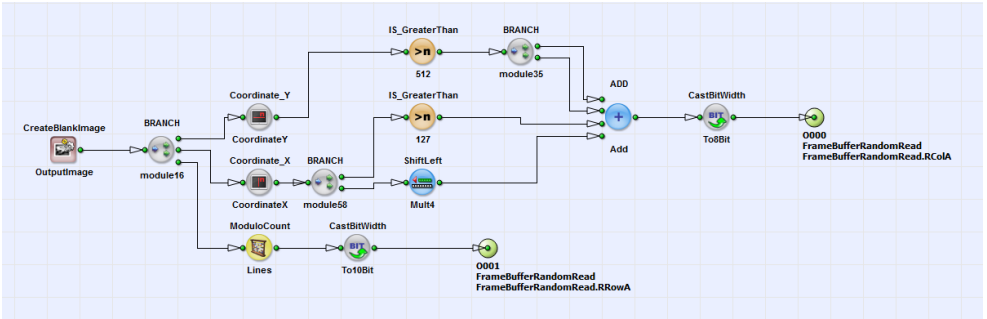



Figure 12.111. Content of the HierarchicalBox **Address** in "TapSorting_2X_2Y.va"

With operators **CoordinateX**, **CoordinateY**, **ModuloCount** and **IS_GreaterThan** the corresponding positions in the input image for each pixel in the output image are evaluated.

For the sorting of the tap geometry of "8X-1Y" and "10X-1Y" in the designs "TapSorting_8X_1Y.va" and "TapSorting_10X_1Y.va" operator **LineMemory** is used. You can see the basic design structure of "TapSorting_8X_1Y.va" in Fig. 12.112. The design structure of "TapSorting_10X_1Y.va" is equivalent.

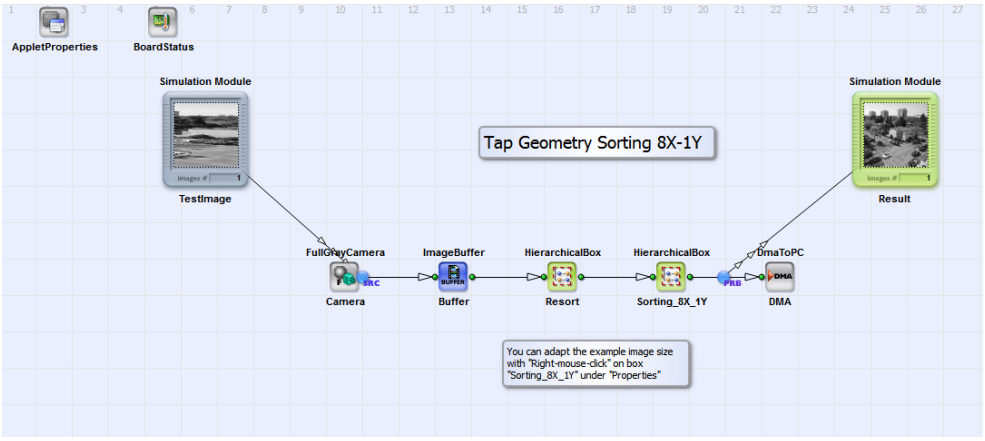



Figure 12.112. Basic design structure of "TapSorting_8X_1Y.va"

The pixels of the eight or ten horizontal taps are rearranged in the HierarchicalBox **Resort**. Eight pixels of the same tap each are merged to one pixel with bit depth 64 bit. The eight or ten taps are merged in parallel. The correct address in the input image for each output pixel is then calculated in the HierachicalBox **Sorting8X1Y** (see Fig. 12.113).

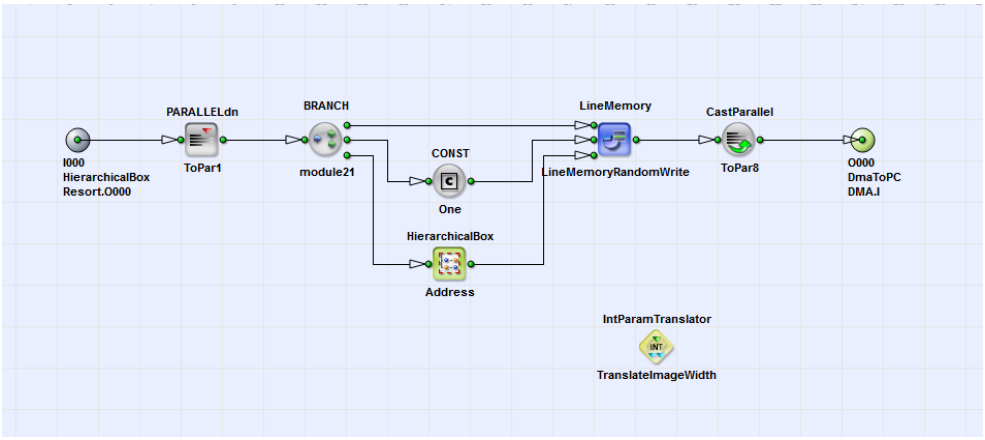



Figure 12.113. Content of the HierarchicalBox **Sorting_8X_1Y**

You can see the the address generation of box **Address** in Fig. 12.114.

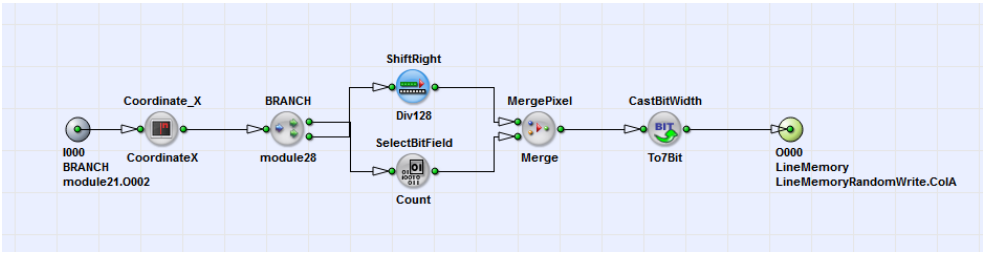


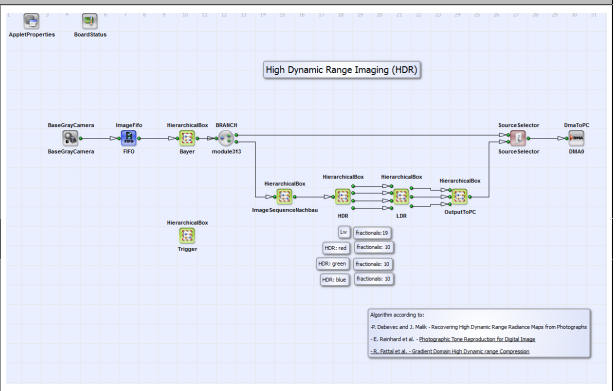

Figure 12.114. Content of the HierarchicalBox **Address** in "TapSorting_8X_1Y.va"

## 12.10. High Dynamic Range and Image Composition

In this section two example algorithms for the creation of a High Dynamic Range image from a sequence of three Standard Dynamic Range (HDR) images with different exposure times are described. One algorithm is a linear approach and the second an algorithm developed by Debevec et. al. [Deb97]. In

addition the tone mapping algorithms for the output of the HDR image on a PC display are described and implemented in the examples. In this section also the example design "ExposureFusion.va" for the image composition according to the method of exposure fusion is introduced. It can be seen as simpler and less resource alternative to High Dynamic Range imaging.

## 12.10.1. High Dynamic Range and Low Dynamic Range Example Using Camera Response Function

| **Brief Description** | |
|---|---|
| **Files:** \examples\Processing \HDR_ImageComposition\HighDynamicRange \HDR_CRC_Bayer.va \examples\Processing\HDR_ImageComposition \HighDynamicRange\HDR_CRC_Color.va \examples\Processing\HDR_ImageComposition \HighDynamicRange\HDR_CRC_Gray.va |   |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>HDR Algorithm According to Debevec and Malik and LDR Algorithm according to Reinhard et al. and Fattal et al.. | |

High Dynamic Range (HDR) images reproduce a high range of luminosity. So both very dark and very bright details are combined in one image. With most photography techniques it is not possible to achieve this high range of luminosity with one single exposure time. So a HDR image is created from images with different exposure times. The HDR image can not be displayed directly on most displays. So a scaled reduction in brightness contrast to a Low Dynamic Range (LDR) image is necessary for displaying the HDR image. This procedure is also called tone mapping. In this VisualApplet example an HDR algorithm according to Debevec and Malik [Deb97] is implemented. The LDR algorithm is according to Reinhard [Rei02] and Fattal [Fat02]. In the following both algorithms will shortly be explained.

### 12.10.1.1. High Dynamic Range Imaging

The pixel value $Z_{x,y}$ in an image i with pixel coordinates $x$ and $y$ can be expressed as a function of the irradiance value $E_{xy}$ and exposure time $dt_i$:

$$Z_{xy,i} = f(E_{xy} \cdot dt_i) \ . \tag{12.32}$$

When we define $g = lnf^{-1}$ it follows:

$$g(Z_{xy,i}) = ln(E_{xy}) + ln(dt_i) \ , \tag{12.33}$$

where we call $g(Z_{xy},i)$ the response curve. It relates the pixel value $Z_{xy,i}$ for exposure time $dt_i$ to the scene irradiance $E_{xy}$. $E_{xy}$ is assumed to be constant for every pixel. Knowing $dt_i$ and $Z_{xy,i}$, $g(Z_xy,i)$ can be calculated with singular value decomposition method [Deb97]. An HDR image can now be created using the following expression [Deb97]:

$$ln(E_{xy}) = \frac{\sum_{i=1}^{P} w(Z_{xy,i}) \cdot [g(Z_{xy,i}) - ln(dt_i)]}{\sum_{i=1}^{P} w(Z_{xy,i}) \ ,} \tag{12.34}$$

where $ln(E_{xy})$ are the logarithmic irradiance values (base $e$) on pixel x,y in the resulting HDR image. For color images $E_{xy}$ are the red, green and blue values ($R_{HDR}$, $G_{HDR}$ and $B_{HDR}$). That is, Equation 12.34 has to be separately calculated for red, green and blue values. $P$ is the number of exposures. In Equation 12.34 $w(Z_{xy,i})$ is a weighting function:

$$w(Z_{xy,i}) = \begin{cases} Z_{xy,i} - Z_{min} \text{ for } z \leq \frac{1}{2}(Z_{min} + Z_{max}) \\ Z_{max} - Z_{xy,i} \text{ for } z > \frac{1}{2}(Z_{min} + Z_{max}) \end{cases} . \tag{12.35}$$

### 12.10.1.2. Low Dynamic Range Imaging

#### 12.10.1.2.1. Bayer and Color Images

To display the HDR image a reduction in brightness contrast is necessary. According to Reinhard et al. [Rei02] the LDR luminance $L_{LDR}$ can be calculated as:

$$L_{LDR} = \frac{L_s}{1 + L_s} . \tag{12.36}$$

Here $L_s$ is the scaled HDR luminance:

$$L_s = a \cdot \frac{L_w}{\bar{L_w}} . \tag{12.37}$$

The parameter $a$ adjusts the brightness of displayed LDR image. Typical values are between 0.09 and 0.72 [Rei02]. The so called relative "world" luminance $L_w$ is calculated from HDR colors: $R_{HDR}$, $G_{HDR}$ and $B_{HDR}$:

$$L_w = 0.2125 \cdot R_{HDR} + 0.7154 \cdot G_{HDR} + 0.0721 \cdot B_{HDR} . \tag{12.38}$$

The mean value $\bar{L_w}$ is in this example calculated as

$$\bar{L_w} = \frac{1}{N} \sum_{1}^{N} L_w . \tag{12.39}$$

According to Fattal et al. [Fat02] the LDR color components $R_{LDR}$, $G_{LDR}$ and $B_{LDR}$ can now be reconstructed as

$$R_{LDR}/G_{LDR}/B_{LDR} = \left( \frac{R_{HDR}/G_{HDR}/B_{HDR}}{L_w} \right)^{0.5} \cdot L_{LDR} . \tag{12.40}$$

#### 12.10.1.2.2. Grayscale Images

The LDR image values $Gray_{LDR}$ for grayscale images for output on display can be calculated as:

$$Gray_{LDR} = \frac{\sqrt{E_{x,y}}}{\sqrt{E_{x,y}} + C} . \tag{12.41}$$

Here $E_{x,y}$ is a result of HDR processing according to Equation 12.34. $C$ is a parameter constant which can be used for adaption of brightness in the output image.

### 12.10.1.3. VisualApplets Design

The HDR/LDR algorithm for filming scenes with very wide luminosity scale with very dark and very bright objects is implemented in VisualApplets for grayscale ("HDR_CRC_Gray.va"), Bayer

("HDR_CRC_Bayer.va") and color ("HDR_CRC_Color.va") camera images. An image is transferred to PC in which every object of the scene is displayed properly. The basic structure of the design "HDR_CRC_Bayer.va" is shown in Fig. 12.115. All examples follow the same principle. For a Bayer pattern raw image the red, green and blue values for every image pixel are calculated in the HierarchicalBox **Bayer** with a **Bayer5x5Linear** operator. This box is only content of the example "HDR_CRC_Bayer.va". In the box **ImageSequence** a sequence of three images is buffered. The three images should have three different exposure times for HDR-LDR processing. The exposure times should be chosen that way that every image pixel is at least in one image of the exposure sequence neither under nor over exposed. You can set these times with the operators **SignalWidth width1** to **width3** in the HierarchicalBox **Trigger** (see Fig. 12.116). Please note that the time scale is system clock ticks of 8 ns. The operator **Generate-Period** has to be set at least to a value greater than the longest time of the operators **SignalWidth width1** to **width3**. Please read in addition the minimum period length for operator **Generate-Period** in your camera manual. The three images are combined using the HDR algorithm according to Debevec and Malik [Deb97] described above. It is implemented in the HierarchicalBox **HDR** (Fig. 12.115). For the grayscale, Bayer and color images the implementation is equivalent. The LDR algorithm is implemented in **LDR**. The implementation for color/Bayer images and grayscale images is different (see sections 12.10.1.2.1 and 12.10.1.2.2). The colors of the resulting RGB image are merged in the HierarchicalBox **OutputToPC**. This box does not exist in the example "HDR_CRC_Gray.va". The operator **SourceSelector** gives the opportunity to select the DMA transport of either the processed HDR-LDR image or the (Bayer demosaiced) camera image.

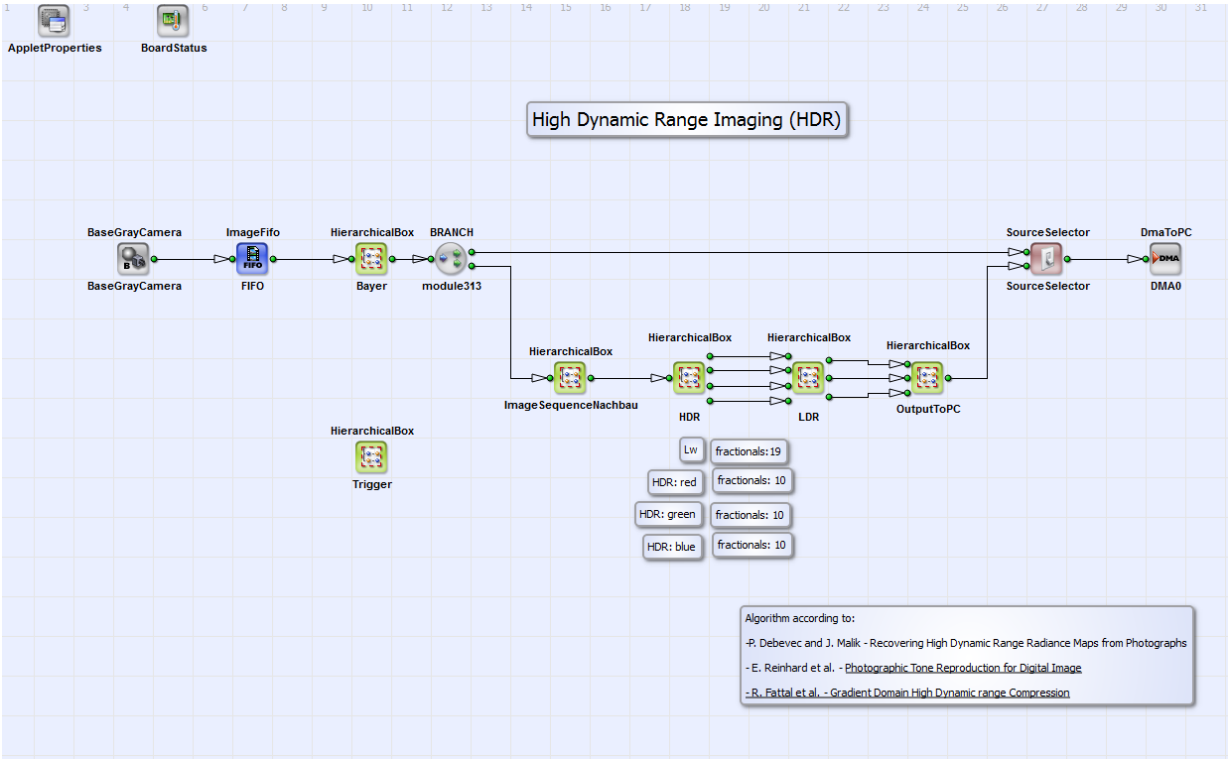



Figure 12.115. Basic design structure

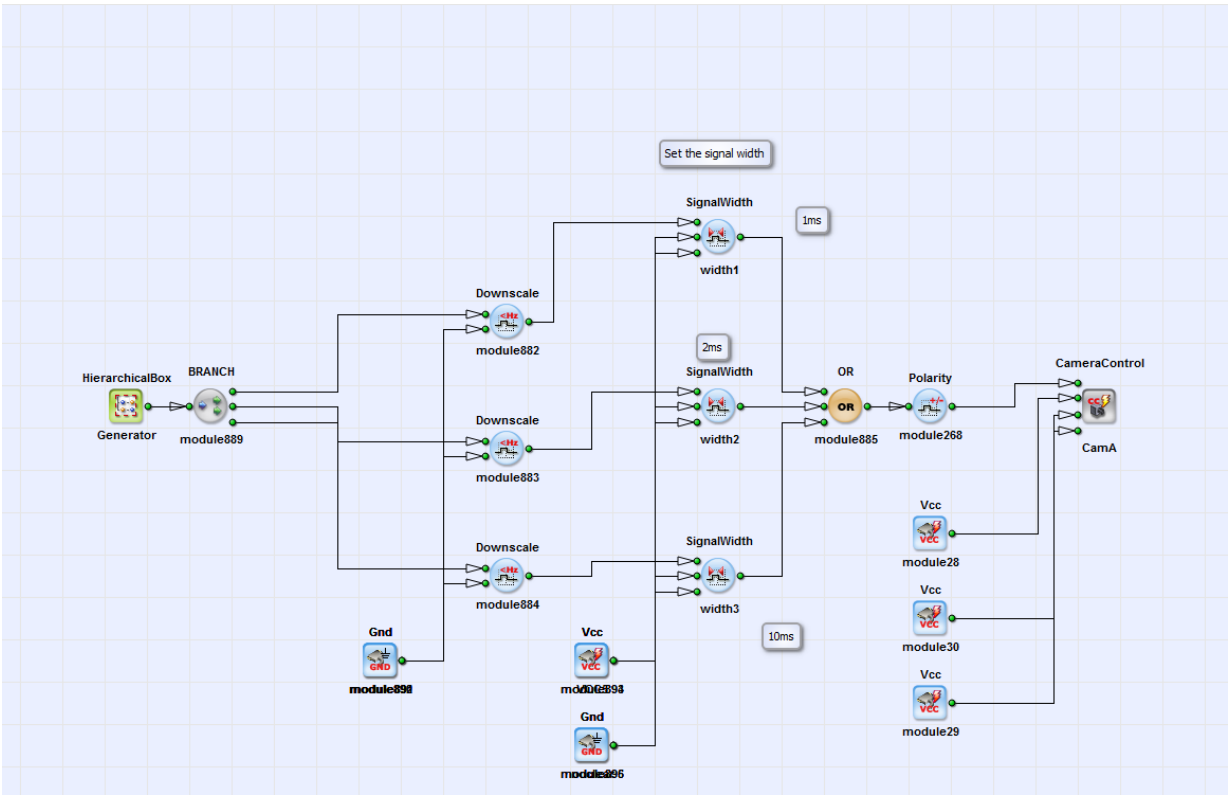



Figure 12.116. Content of box **Trigger**

In Fig. 12.117 you can see the content of the HierarchicalBox **HDR**. For the three images of the buffered image sequence the summands of the nominator $w(Z_{xy,i}) \cdot [g(Z_{xy,i}) - ln(dt_i)]$ and denominator $w(Z_{xy,i})$ of Equation 12.34 are calculated for images one to three (HierarchicalBox **Image1** to **Image3**) for the colors red, green and blue separately.

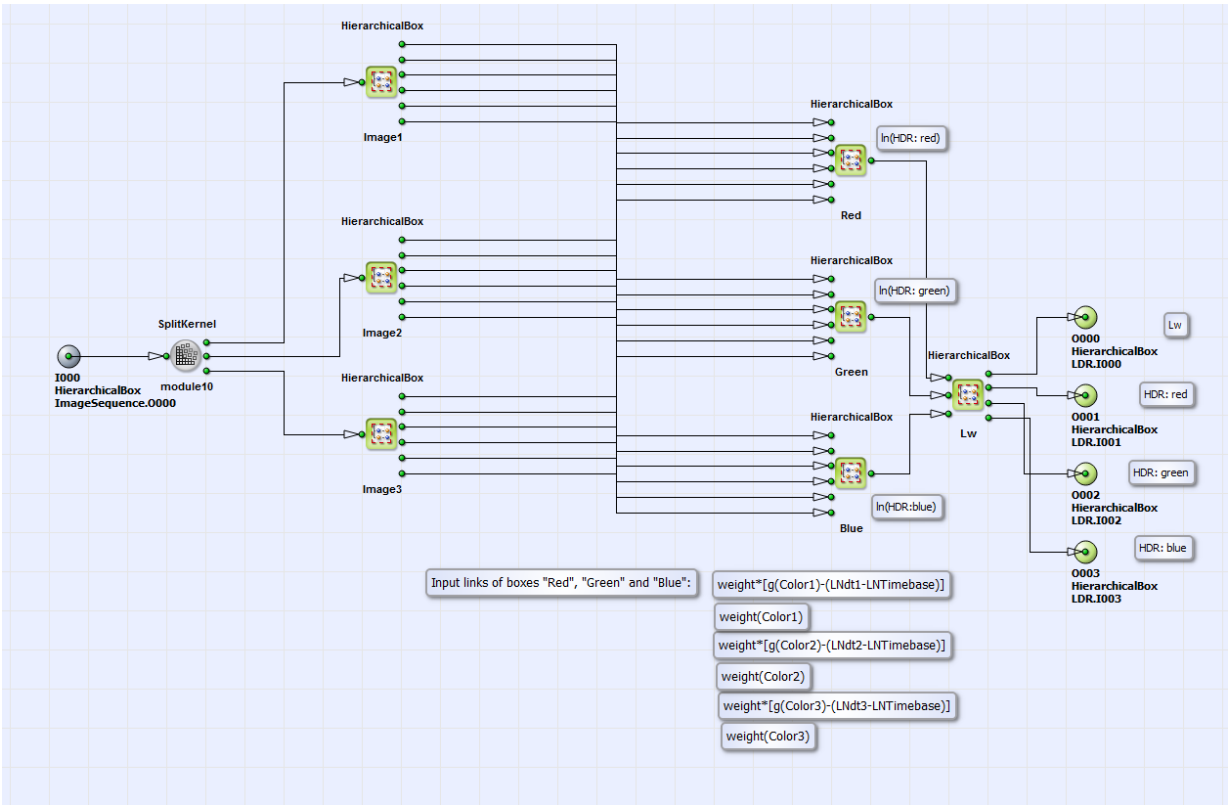



Figure 12.117. Content of box **HDR**

In Fig. 12.118 you can see this calculation for the color red for image 1 (in the HierarchicalBox **Red** in the box **Image1**) as an example. All colors for all images are processed the same way. The logarithmic values of the camera response curve (CRC) are assigned to the (red/green/blue) pixel values with a lookup table in the HierarchicalBox **over127** if pixel value is higher than 127 or in box **underequal127** if the pixel value is less equal than 127. The content of these lookup tables **gO127** and **g_ueq_127** for the colors red, green and blue can be calculated with the Matlab program modules "HDR_CRC.m", "sample.m" and "gsolve.m" (under \examples\Processing\Advanced\HighDynamicRange\), which are based on the code of Debevec and Malik [Deb97] and partially on the example of M. Eitz [Eit07]. Please read the short manual in "HDR_CRC.m" for further instructions how to use the code. With the **CONST** operator **LNdt1** (and analog **LNdt2** and **LNdt3** for images 2 and 3) the logarithmic (base: e) value of the exposure time can be set. Please note, that this value has to be multiplied by $2^{16}$ because of 16 fractional bits. The corresponding time base can be chosen by the parameter **LNTimeBase**. Values of $ln(1) \cdot 2^{16} (= 0\ !)$, $ln(1000) \cdot 2^{16}$ or $ln(1000000) \cdot 2^{16}$ set the time scale to seconds, milliseconds or microseconds. In the lookup table **weight** in Fig. 12.118 the weighting function of Equation 12.35 is implemented. With the Matlab program modules "HDR_CRC.m", "sample.m" and "Weight.m" the weighting table "Weights.txt" can be created. Please read also the short manual in "HDR_CRC.m" (or "HDR_CRC_Gray.m" for grayscale images) for further instructions. Final outputs of the HierarchicalBox **Red** (and analog **Green** and **Blue**) in box **Image1** (and analog **Image2** and **Image3**) are then $w(Z_{xy,i}) \cdot [g(Z_{xy,i}) - ln(dt_i)]$ and $w(Z_{xy,i})$.

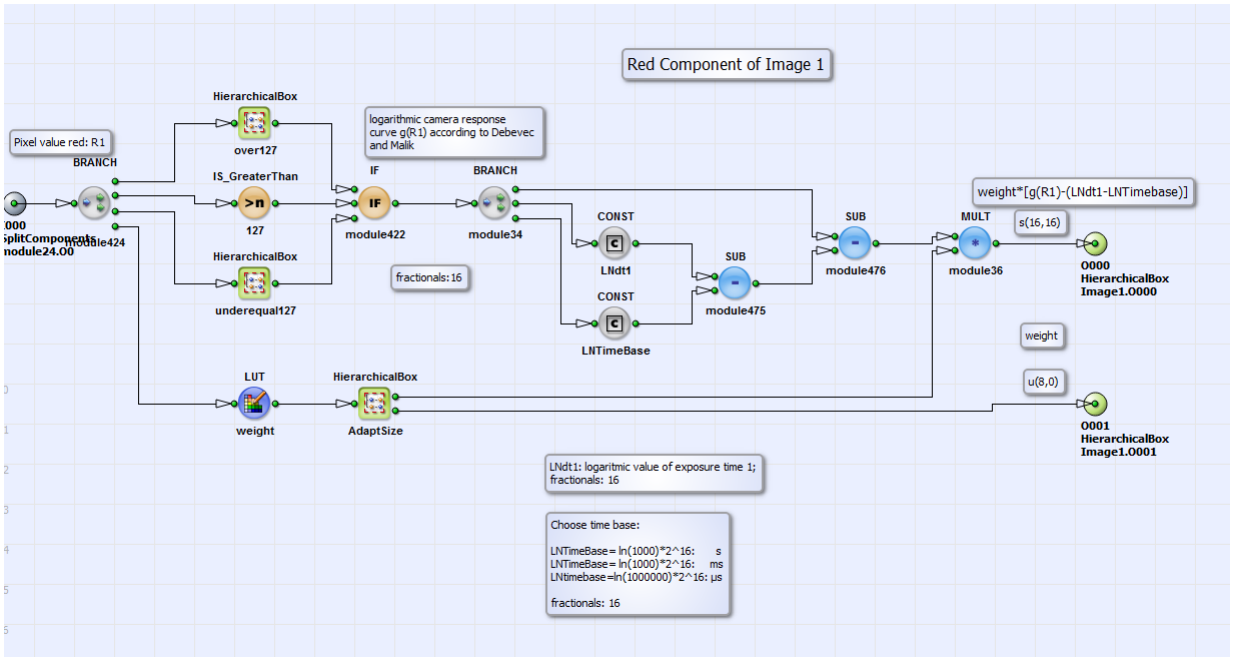



Figure 12.118. Content of component **Red** in **Image1**

Coming back to Fig. 12.117 as content of box **HDR**. The calculation of the logarithmic values of the HDR color components $ln(Red_{HDR})$, $ln(Green_{HDR})$ and $ln(Blue_{HDR})$ according to Equation 12.34 is performed in the boxes **Red**, **Green** and **Blue**. A summation over all components of nominator $w(Z_{xy,i}) \cdot [g(Z_{xy,i}) - ln(dt_i)]$ and denominator $w(Z_{xy,i})$ values and final division is implemented here (see color red in Fig. 12.119 for example).

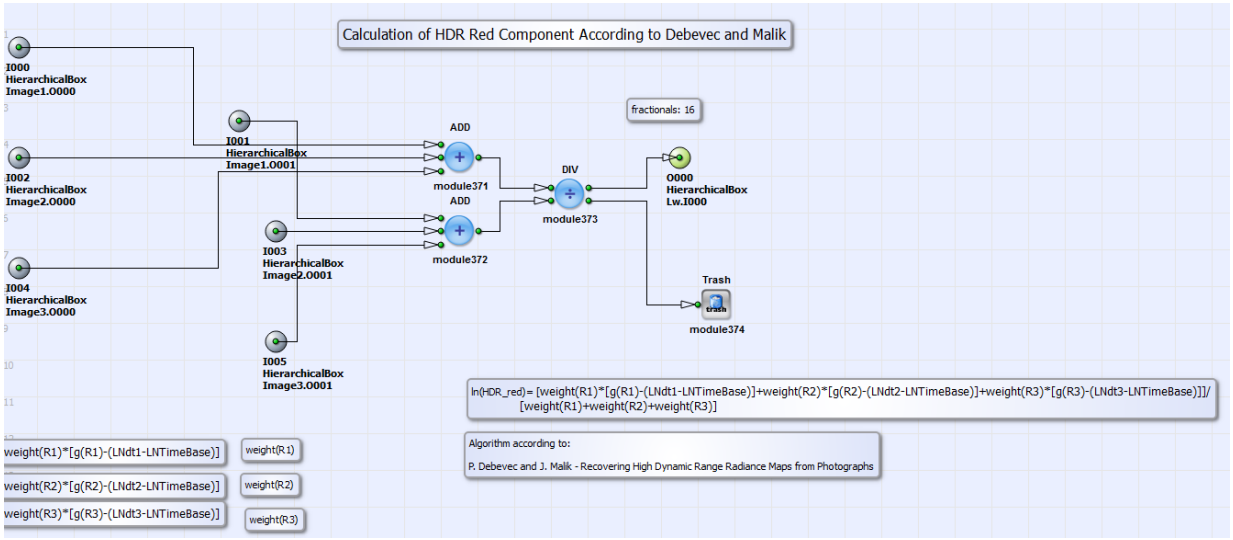



Figure 12.119. Content of box **Red** under **HDR**

The box **Lw** in Fig. 12.117 contains the calculation of the HDR color values $Red_{HDR}$, $Green_{HDR}$ and $Blue_{HDR}$ from their logarithmic values. The exponential function is implemented with lookup tables. The relative "world" luminance "Lw" is then calculated from the HDR color components according to Equation 12.38. Its mean value according to Equation 12.39 is calculated in box **Lw_d** in the module **LDR** (see basic design structure Fig. 12.115). The whole content of **LDR** is shown in Fig. 12.120.

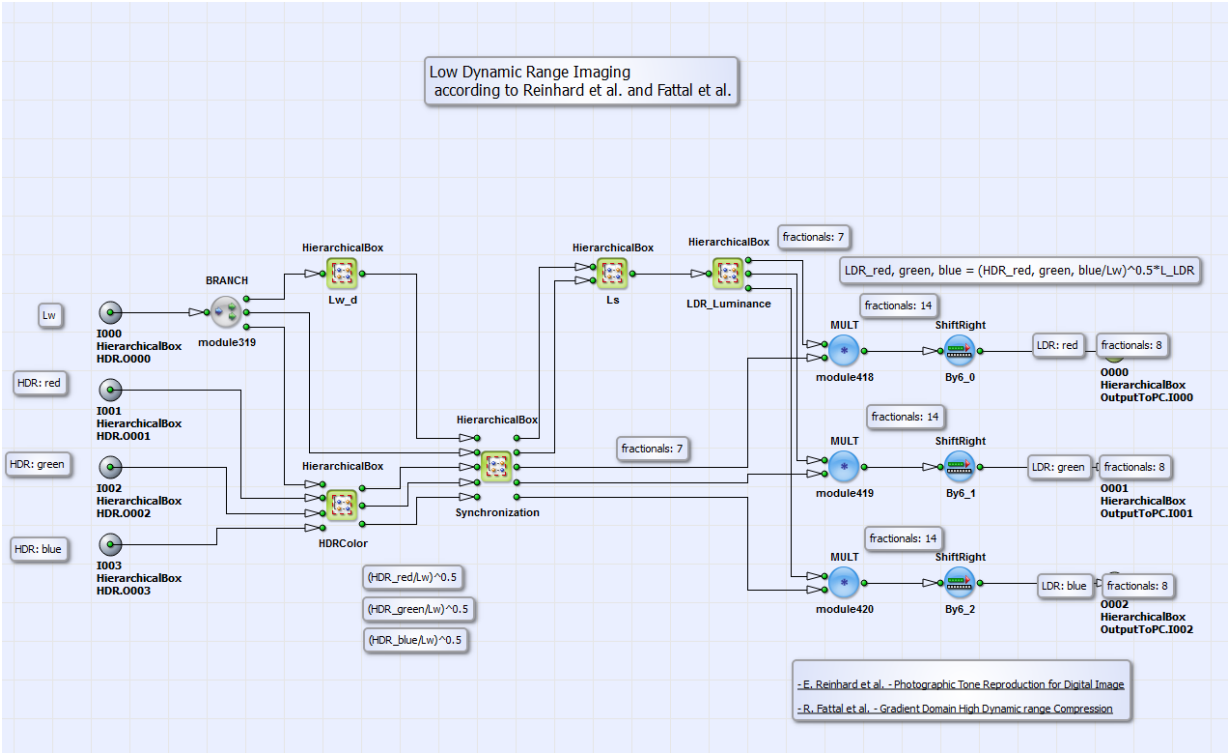



Figure 12.120. Content of box **LDR** in the designs "HDR_CRC_Bayer.va" and "HDR_CRC_Color.va"

The module **Ls** contains the calculation of the scaled luminance according to Equation 12.37 as division of relative "world" luminance and its mean value multiplied by a "brightness parameter" **a**. By changing this parameter value you can change the brightness of the final processed image. You can find appropriate values for this parameter in the comment box in the VA design or in case of deeper interest in [Rei02]. **LDR_Luminance** calculates the LDR luminance according to Equation 12.36. This value is finally multiplied with the outputs of the HierarchicalBox **HDRColor** $(Red_{HDR}/Lw)^{0.5}$, $(Green_{HDR}/Lw)^{0.5}$ and $(Blue_{HDR}/Lw)^{0.5}$ for color components red, green and blue separately. The results are then the LDR color components red, green and blue [Fat02] of the processed output image. The content of HierarchicalBox **LDR** for grayscale images is shown in Fig. 12.121.

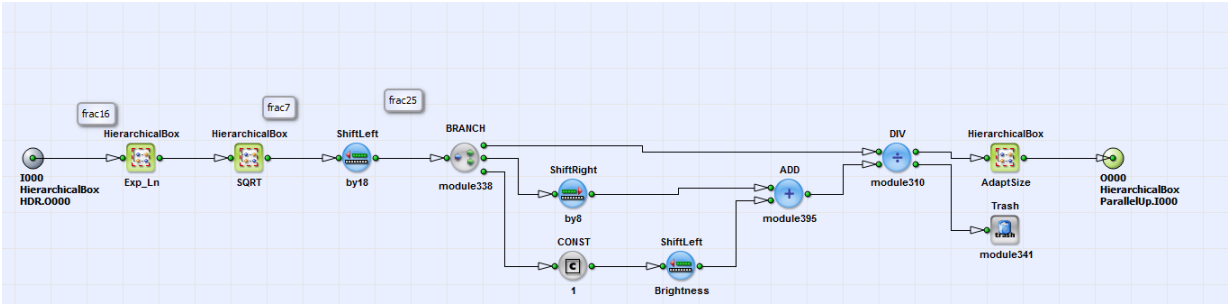
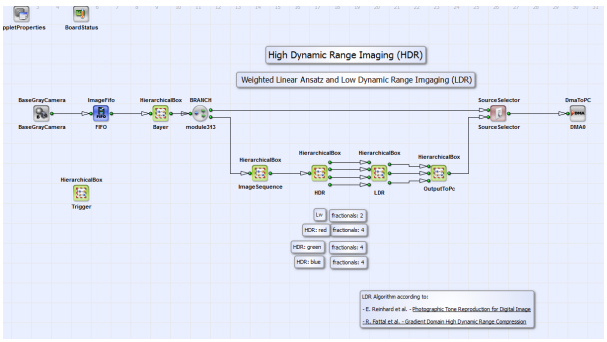



Figure 12.121. Content of box **LDR** in the design "HDR_CRC_Gray.va"

In the HierarchicalBox **Exp_Ln** $E_{x,y}$ is calculated from its logarithmic value $E_{x,y}$ as result of HDR processing (Equation 12.34). In the box **SQRT** the square root is extracted according to the algorithm in section 12.10.1.2.2. The result is divided by itself plus $1 \cdot 2^n$. The value of $n$ and with it the brightness of the output image can be set with the operator **ShiftLeft_Brightness**.

## 12.10.2. High Dynamic Range and Low Dynamic Range Example with a Weighted Linear Ansatz

| Brief Description | |
|---|---|
| **Files:** \examples\Processing \HDR_ImageComposition\HighDynamicRange \HDR_linearW_Bayer_3_Base.va \examples\Processing \HDR_ImageComposition\HighDynamicRange \HDR_linearW_Color_3_Base.va \examples\Processing \HDR_ImageComposition\HighDynamicRange \HDR_linearW_Gray_3_Base.va \examples\Processing \HDR_ImageComposition\HighDynamicRange \HDR_linearW_Bayer_3_Full.va \examples\Processing \HDR_ImageComposition\HighDynamicRange \HDR_linearW_Color_3_Full.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Weighted Linear HDR Algorithm and LDR Algorithm according to Reinhard et al. and Fattal et al.. | |

This High Dynamic Range (HDR) and Low Dynamic Range (LDR) VisualApplets (VA) design is similar to the one described in Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'. The difference is the HDR algorithm. This will be described in the following subsection. There exist several variations of this example: They are designed for a marathon VCL board in base and full configuration for a RGB, a Bayer pattern or a grayscale camera.

### 12.10.2.1. High Dynamic Range Imaging With Weighted Linear Ansatz

The exposure $X$ can be written as inverse function of Equation 12.32

$$X = f^{-1}(Z_{xy,i}) = E_{xy} \cdot dt_i \ . \tag{12.42}$$

If you can suppose the exposure to be a linear function you can construct the HDR color components with:

$$R_{HDR}/G_{HDR}/B_{HDR} = \frac{\sum_{i=1}^{P} w(Z_{xy,i}) \cdot [Z_{xy,i}/dt_i]}{\sum_{i=1}^{P} w(Z_{xy,i})} \tag{12.43}$$

instead of with Equation 12.34. Here $w(Z_{xy,i})$ is analog to Equation 12.35 a weighting function with which under and over exposed pixel values are less emphasized. The LDR components in this example are calculated analog to the LDR algorithms in 12.10.1.2.1 and 12.10.1.2.2.

### 12.10.2.2. VisualApplets Design

The example is suitable for filming scenes with very wide luminosity scale with very dark and very bright objects. An image is transferred to PC in which every object of the scene is displayed properly. The basic structure of the design is shown in Fig. 12.122 (here for a Bayer pattern camera in base configuration). The red, green and blue values for every image pixel are calculated in the HierarchicalBox **Bayer** with a **Bayer5x5Linear** operator. In the box **ImageSequence** a sequence of three images is buffered. The three images should have three different exposure times for HDR-LDR processing. The exposure times should be chosen that way that every image pixel is at least in one image of the exposure sequence neither under nor over exposed. You can set these times with the operators **SignalWidth width1** to **width3** in the HierarchicalBox **Trigger** (see Fig. 12.116). Please note that the time scale is system clock ticks of 8 ns. The operator **Generate-Period** has to be set at least to a value greater

than the longest time of the operators **SignalWidth width1** to **width3**. Please read in addition the minimum period length for operator **Generate-Period** in your camera manual. The three images are combined using the weighted linear HDR algorithm described in Equation 12.43. It is implemented in the HierarchicalBox **HDR** (Fig. 12.122). The LDR algorithm according to Reinhard et al. [Rei02] and Fattal et al.[Fat02] is implemented in **LDR**. The colors of the resulting RGB image are merged in the HierarchicalBox **OutputToPC**. In the design "HDR_linearW_Gray_3_Base.va" this box does not exist. The operator **SourceSelector** gives the opportunity to select the DMA transport of either the processed HDR-LDR image or the (Bayer demosaiced) camera image. Since the design structure and calculations are analog to the ones of the example in section 12.10.1 we will just concentrate on the explanation of the content of **HDR** in the following. But please note that in some details like fractional bits the designs might also differ. You find the corresponding hints in the comment boxes in the examples.

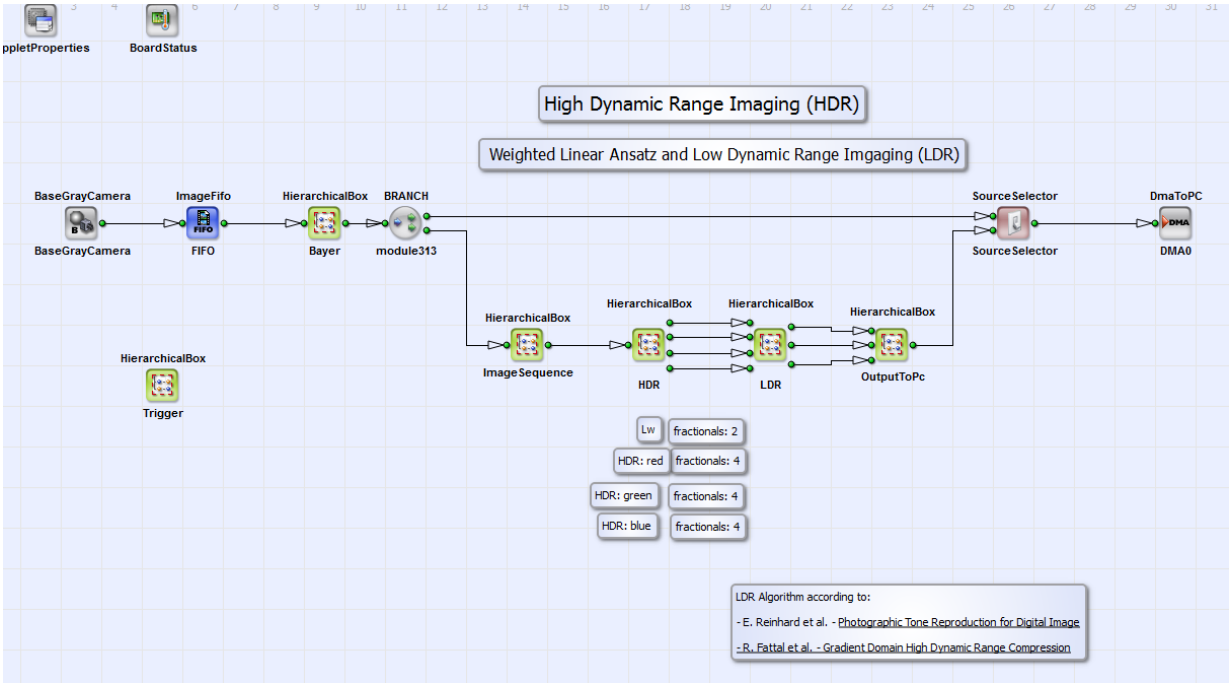



Figure 12.122. Basic design structure

In Fig. 12.123 you can see the content of the HierarchicalBox **HDR**. For the three images of the buffered image sequence the summands of the nominator $w(Z_{xy,i}) \cdot [Z_{xy,i}/dt_i]$ and denominator $w(Z_{xy,i})$ of Equation 12.43 are calculated for images one to three (HierarchicalBox **Image1** to **Image3**) for the colors red, green and blue separately. For grayscale images in the design "HDR_linearW_Gray_3_Base.va" the calculation is done for the grayscale component only .

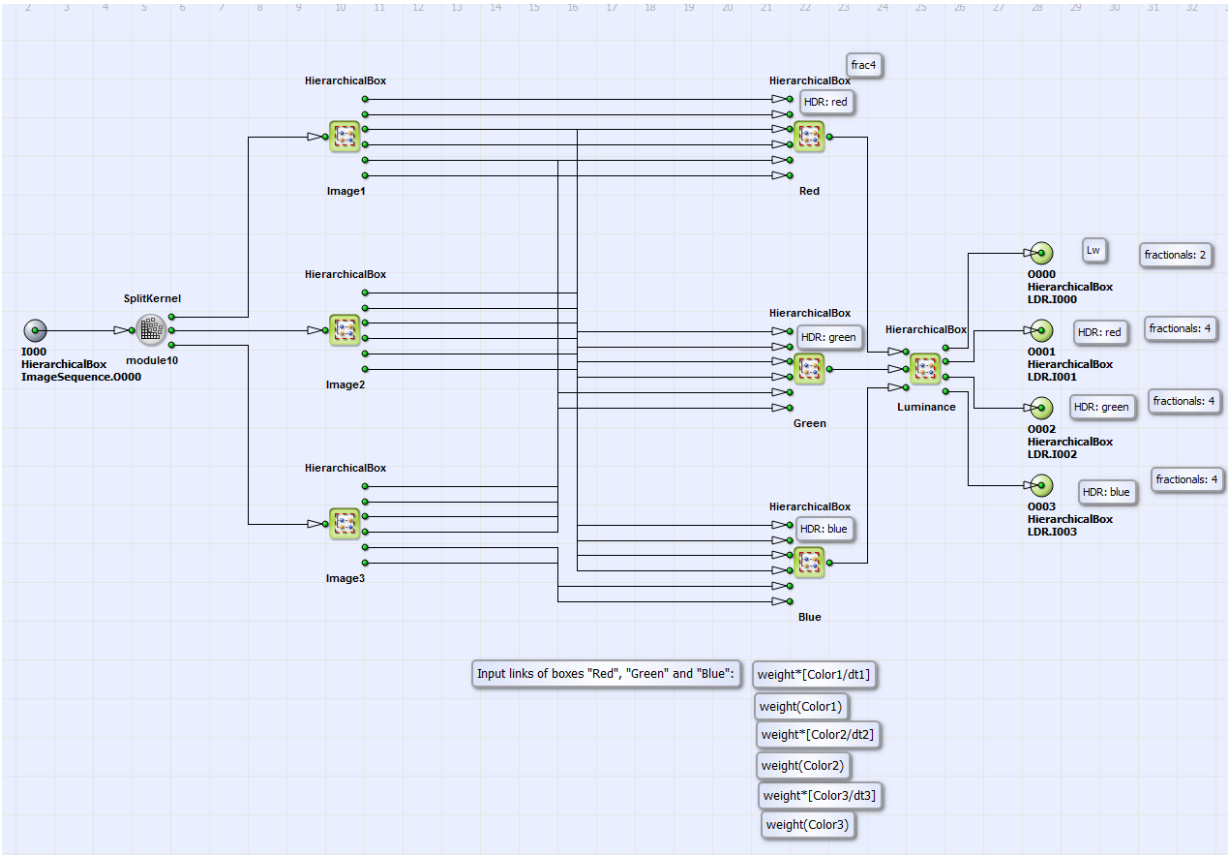



Figure 12.123. Content of box **HDR**

The calculation of these summands is shown in Fig. 12.124 for color red in **Image1** as example. The calculation for the colors green and blue and images 2 and 3 are analog. The pixel value in Fig. 12.124 is divided by the exposure time which can be set by the value of the parameter **Const_dt1** (or analog **Const_dt2** and **Const_dt3**). The time base can be seconds, milli- or microseconds and can be set by the parameter **TimeBase** in the boxes **Red**, **Green** and **Blue** in the box **HDR** (see Fig. 12.123 and Fig. 12.125). The result of $Z_{xy,i}/dt_i$ is multiplied with a weighting function implemented as lookup table **LUT_weight**. The table "WeightsLinear.txt" can be created using the Matlab program modules "WeightLinear.m", "sample.m" (or "sample_gray.m" for grayscale images) and "HDR_CRC_m" (or "HDR_CRC_Gray.m" for grayscale images) (under \examples \Processing\Advanced\HighDynamicRange\). Please read also the short manual in "HDR_CRC.m" (or "HDR_CRC_Gray.m" for grayscale images) for further instructions. The outputs of the component shown in Fig. 12.124 are then the summands of Equation 12.43: $w(Z_{xy,i}) \cdot [Z_{xy,i}/dt_i]$ and $w(Z_{xy,i})$.

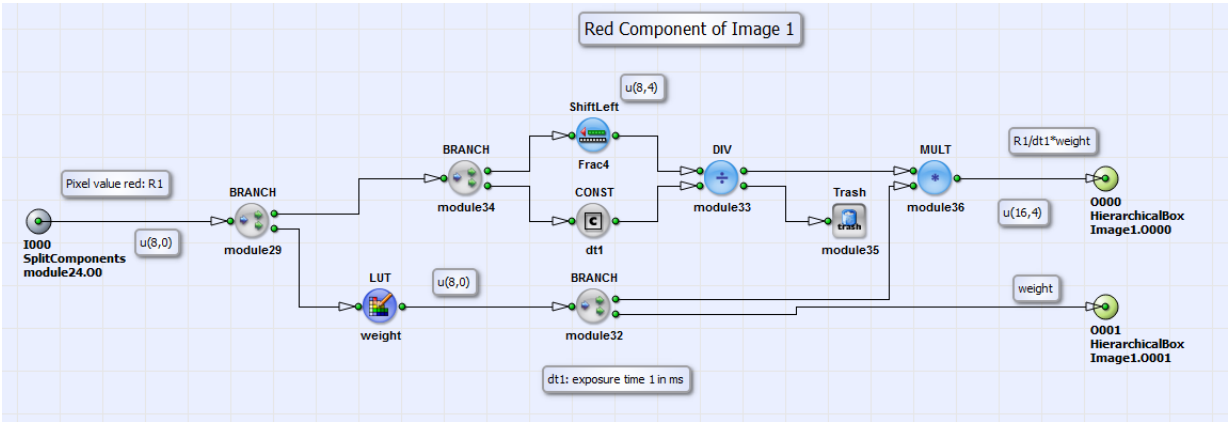



Figure 12.124. Content of **Red** in box **Image1**

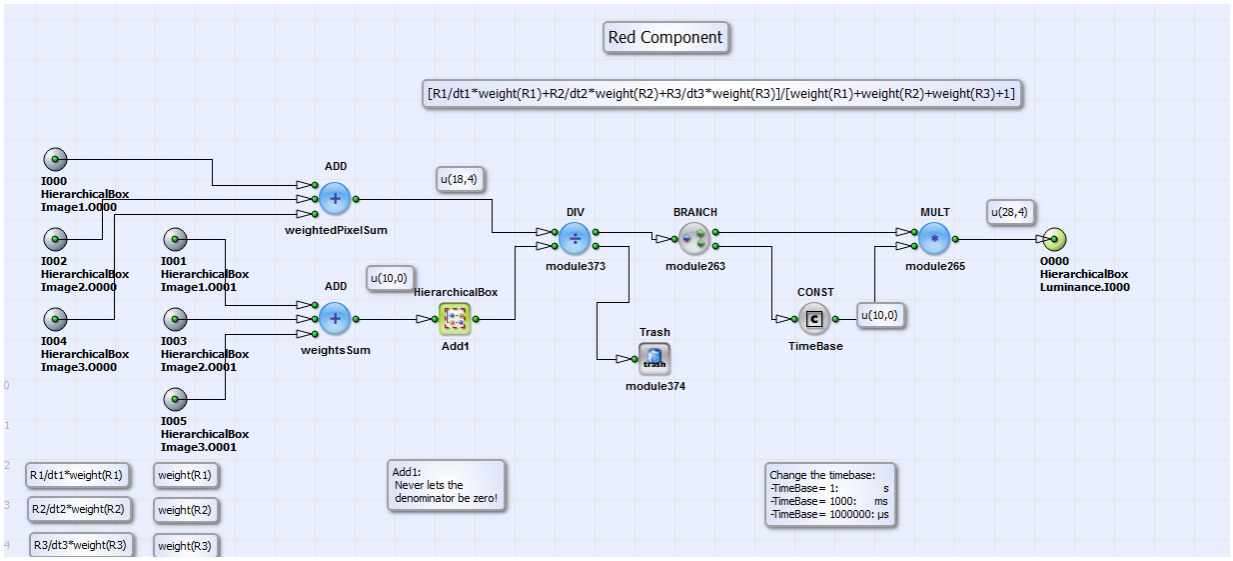
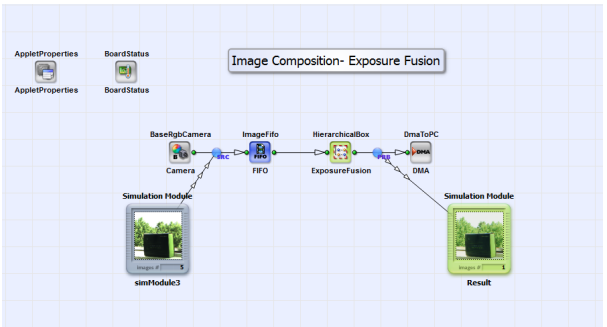



Figure 12.125. Content of **Red** in box **HDR**

These summands are summed up and divided according to Equation 12.43 in the HierarchicalBoxes **Red**, **Green** and **Blue**. See for example box **Red** in Fig. 12.125.

The calculation of the luminance $L_w$ in the module **Luminance**, of the scaled luminance $L_s$, of the LDR luminance $L_{LDR}$ and the final calculation of the LDR color components $R_{LDR}$, $G_{LDR}$ and $B_{LDR}$ or $Gray_{LDR}$ is analog to the implementation in section Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'. Please refer for further reading the corresponding parts in section 12.10.1 or the comment boxes in the VA design.

## 12.10.3. Image Composition Using Exposure Fusion

| **Brief Description** | |
|---|---|
| **Files:** \examples\Processing \HDR_ImageComposition\ExposureFusion \ExposureFusion.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>An image composition from up to 16 images with different exposure times to one image showing details and contrasts over complete image without over- and under exposed pixels. This example is a simple alternative to High Dynamic Range Imaging. | |



Exposure fusion combines images with different exposure times to one image. In this resulting image under- and over exposed pixels are prevented and details are fully conserved. The aim of exposure fusion is similar to High Dynamic Range Imaging (see section 12.10.1 and 12.10.2), but in contrast to this method, the dynamic range of luminosity of the acquired images is not extended. Exposure fusion simply takes the best parts of the images of the sequence and combines them in one result image. Exposure fusion is easier to handle for the user, as he does not need to know the exposure times of the images acquired. The quality of the resulting image is slightly worse than with the algorithms of HDRI. In the following the exposure fusion algorithm, which is implemented in "ExposureFusion.va" is introduced.

### 12.10.3.1. Theory of Exposure Fusion

The exposure fusion algorithm according to [Mer07] combines images of a sequence with length N with a weighted blending to a result image $R(x,y)$:

$$R(x,y) = \sum_{k=1}^{N} \hat{W}_k(x,y) \cdot I_k(x,y) \tag{12.44}$$

Here $\hat{W}_k(x,y)$ is the weighting for each color component red, green and blue $I(x,y)$ of a pixel at position $x,y$ in the k-th image of the sequence with

$$\hat{W}_k(x,y) = \frac{3 \cdot W_k(x,y)}{\sum_k^N \sum_{red,green,blue} W_k(x,y)} \tag{12.45}$$

.

$W_k(x,y)$ is a weighting function, which is dependent on the pixel value at position $x,y$ and is a measure for the well-exposedness of this pixel. In "ExposureFusion.va" $W_k(x,y)$ is a linear function up to the pixel value of 127. For pixel values from 128 to 255, $W_k(x,y)$ is set to 127. In dependence on his requirements the user easily can change this weighting function, as described in section 12.10.3.2 e.g. according to [Mer07]. Quality measures like contrast or saturation (see [Mer07]) are not implemented in this reference design. In eq. 12.45 the sum over the color components red, green and blue in the denominator prevents, that a pixel $R(x,y)$ in the result image looses the color information of the original images.

### 12.10.3.2. Implementation in VisualApplets

In Fig. 12.126 the basic design structure of "ExposureFusion.va" is shown.

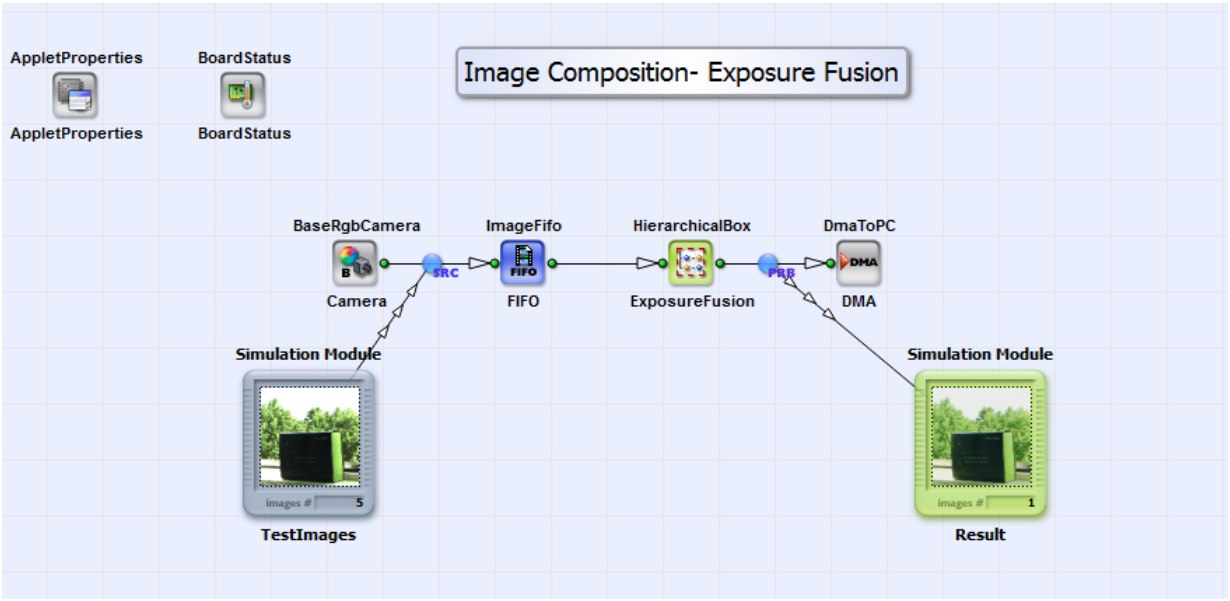



Figure 12.126. Basic design structure of "ExposureFusion.va"

The images of a sequence from an rgb camera in Camera link base configuration are combined to one result image according to the algorithm in eq. 12.44 in the HierachicalBox **ExposureFusion**. For simulation purpose please load up to 16 images of maximum dimensions of 1024x1024 pixels to the simulation source **TestImages**. The order of the images with respect to their exposure times is not of importance! With "right-mouse-click" on box **ExposureFusion** you can set the image sequence length and the image dimensions of the input images. The maximum number of images is 16 and the

maximum image dimensions are 1024x1024 pixels. The output image is sent via DMA to PC. In Fig. 12.127 the content of the HierarchicalBox **ExposureFusion** is shown.

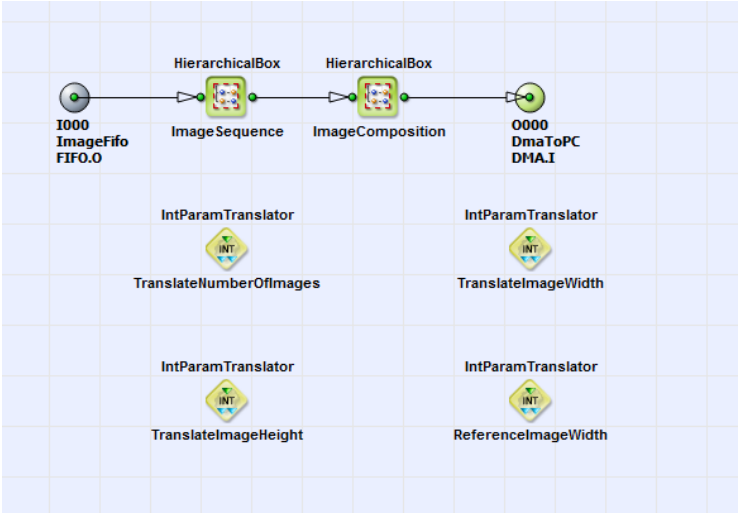



Figure 12.127. Content of HierarchicalBox **ExposureFusion**

In the box **ImageSequence** the images from the sequence are combined to one large image. The pixels at position $x,y$ from the images are positioned as neighbors. In the box **ImageComposition** the exposure fusion algorithm according to eq. 12.44 is implemented. You can see its content in Fig. 12.128.

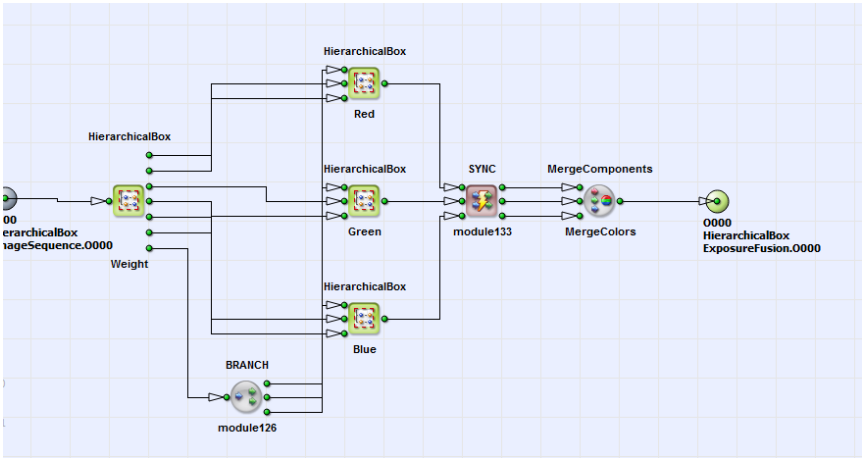



Figure 12.128. Content of HierarchicalBox **ImageComposition**

In box **Weight** the weighting function $W_k(x,y)$ (see. eq. 12.45) in dependence of the pixel value is implemented for each color component. You can see the content of box **Weight** in Fig. 12.129.

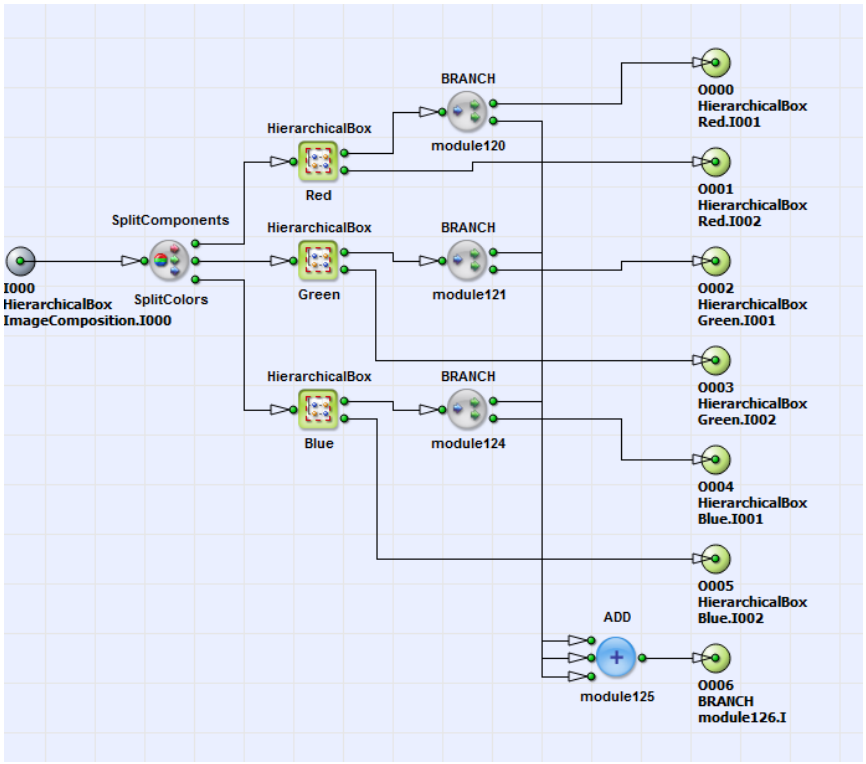



Figure 12.129. Content of HierarchicalBox **Weight**

The weighting functions for each color component from boxes **Red**, **Green** and **Blue** are summed up, according to eq. 12.45. In Fig. 12.130 you can see the weighting function $W_k(x,y)$ for the color red.

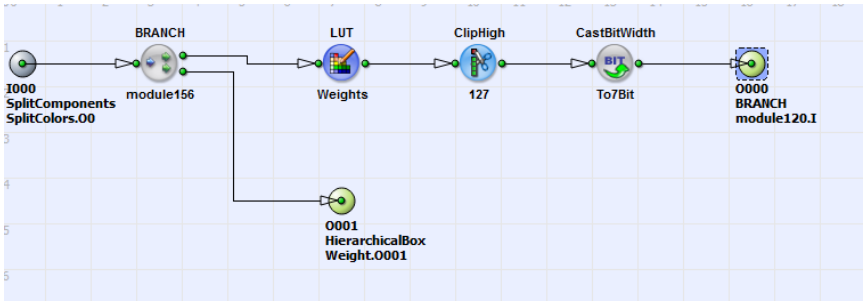



Figure 12.130. Content of HierarchicalBox **Red** in box **Weight**

Up to a pixel value of 127 the function is linear. From pixels values 128 to 255, $W_k(x,y)$ is set to 127. The lookup table operator **LUT_Weights** allows the user to adapt the weighting function easily to his requirements. Also changing the parameters for the **ClipHigh** and **CastBitWidth** operators (or even deleting them) gives the user this possibility. Coming back to the content of box **ImageComposition**. Here according to eq.12.44 the pixel values $I(x,y)$ for the color components red, green and blue are multiplied with the weighting function $W_k(x,y)$ and with a constant **Const** of value 3 in the boxes **Red**, **Green** and **Blue**. You can see the content of box **Red** in Fig. 12.131.

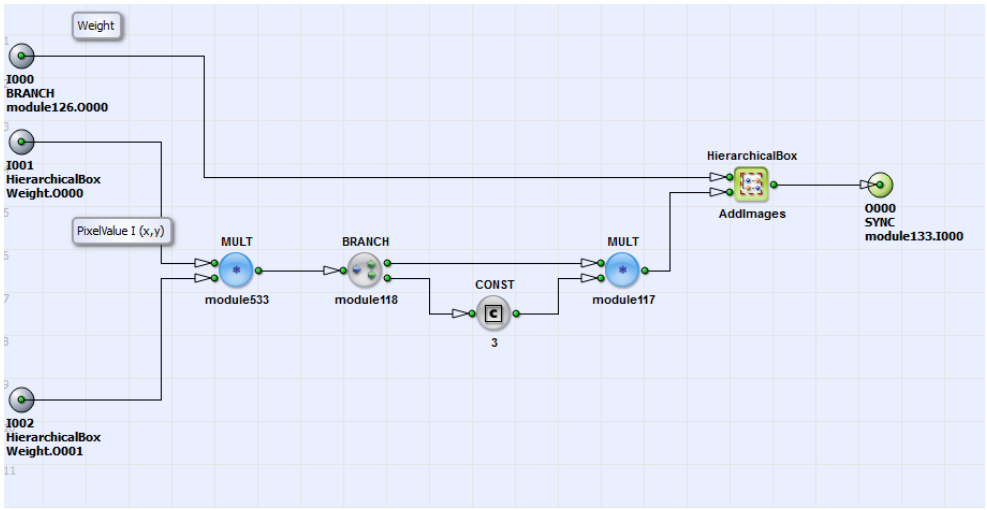



Figure 12.131. Content of HierarchicalBox **Red** in box **ImageComposition**

Here $\sum_{red,green,blue} W_k(x,y)$ is summed up for all images. $3 \cdot I_k(x,y) \cdot W_k(x,y)$ is then divided by $\sum_{k}^{N} \sum_{red,green,blue} W_k(x,y)$ according to eq. 12.45. Finally the color components are merged together (see Fig. 12.128) and the output image is sent to PC. In Fig. 12.132 you can see 5 example input images with the result image in Fig. 12.133. The result image has no under- and over exposed pixels and details are conserved.

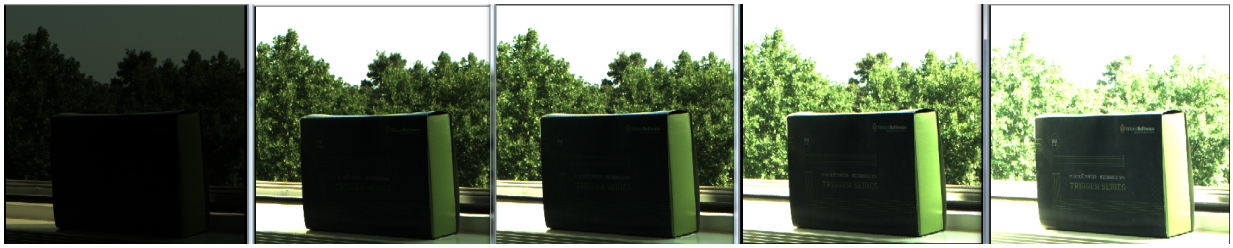


Figure 12.132. Example input images with different exposure times

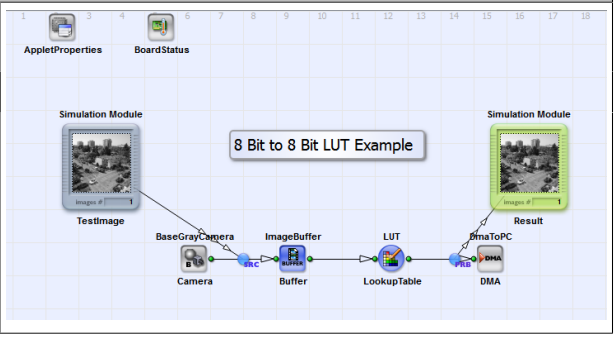



Figure 12.133. Result image of the 5 example input images after exposure fusion

## 12.11. Lookup Tables

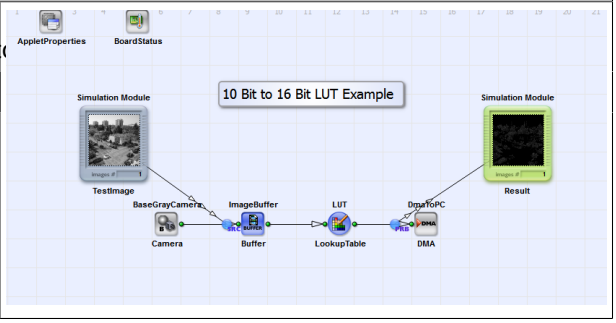
The VisualApplets examples in the following subsections introduce the usage of the lookup-table operators **LUT** and **KneeLUT**. Four examples are provided for different input bit depths for a grayscale image and also one example for using a **LUT** for a RGB image.

### 12.11.1. Lookup Table 8 Bit

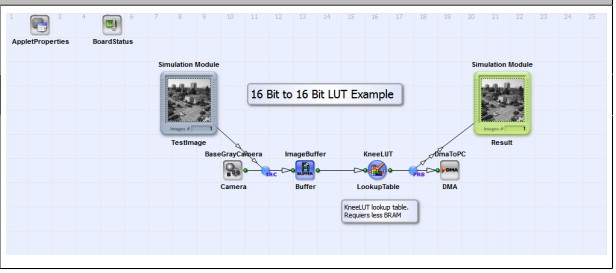
| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\LookupTable \LUT_BaseAreaGray8\LUT_BaseAreaGray8.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows the use of a 8 Bit to 8 Bit lookup table. | |



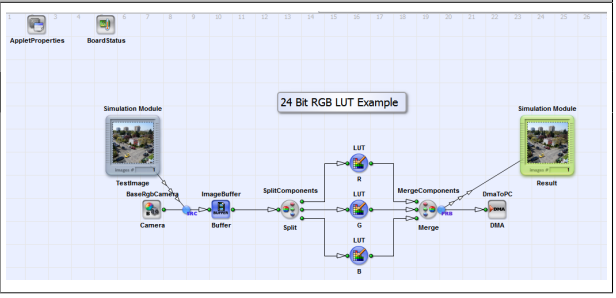
### 12.11.2. Lookup Table 10 to 16 Bit

| Brief Description | |
|---|---|
| **File:** \examples\Processing\LookupTable \LUT_BaseAreaGray10to16\LUT_BaseAreaGray10t |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows the use of a lookup table with 10 bit input and 16 bit output. | |

### 12.11.3. Knee-Lookup Table 16 Bit

| Brief Description | |
|---|---|
| **File:** \examples\Processing\LookupTable \LUT_BaseAreaGray16\LUT_BaseAreaGray16.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Shows the use of a lookup table for 16 Bit input and output data. For 16 bit a Knee LUT has to be used due to the limited block RAM resources. | |

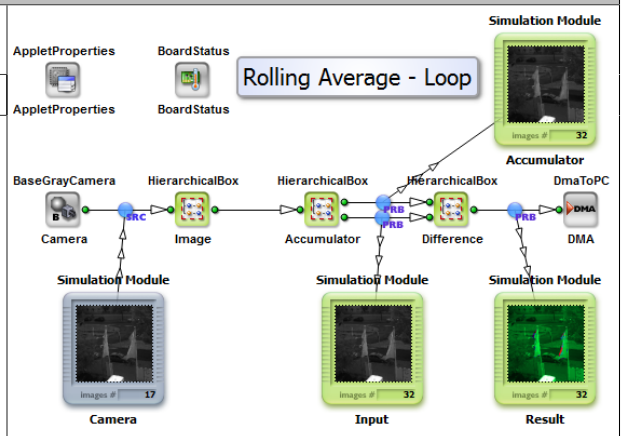
### 12.11.4. Knee-Lookup Table 24 Bit Color

| Brief Description | |
|---|---|
| **File:** \examples\Processing\LookupTable \LUT_BaseAreaRGB24\LUT_BaseAreaRGB24.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>In this example three lookup tables are used for RGB color correction. | |

## 12.12. Loop

This section contains two examples using loops. One calculates a rolling average, the second restores the three dimensional information of an object from a sequence of partially focused 2 D images and creates a completely focused image.

### 12.12.1. A rolling average is applied on a dynamic number of images

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Loop\RollingAverage\rollingAverage_Loop.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Implementation of a rolling average on a running image stream without introducing a frame delay. The average value of the last N monochrome images is calculated and used to visualize the resulting difference to the current one in color. The number of images being used for building the average value is set to 128, but can be modified. | |

This design will show how a rolling average will work. But the central element is a loop that will enable much more applications and machine vision operations. In order to understand how a loop works and how it can become implemented, this example is a perfect starting point.

### 12.12.1.1. Algorithm

A mean value for each single pixel is calculated and normalized to the incoming value range of 8 bit. For this each received frame is summed up with its predecessors and divided by the number of received frames. If the length of the rolling average sequence is reached the sum is reduced by the stored and delayed frame. By this an output image is already generated starting from the first image.

In the design two conditions define the length of the rolling average sequence. Both values are called N_rolling and need to be set to the same value. One is handling the delay store process of substraction images, the other the pixel value normalization.

### 12.12.1.2. Used Loop

The operators RxImageLink and TxImageLink are used to enable a loop handling. The not normalized average sum of the incoming image sequence is handled in one loop. A second loop handling is required for storing the incoming frames for the later substraction of it values. Each loop needs to be synchronous to the incoming camera frames. This induces a SYNC operator where a start condition needs to be met. The start condition itself is handled by InsertImage where the first run introduces a single minimum frame. The used SYNC maximizes the frame dimensions to the needed value.

Since the average handling needs a certain number of frames being stored the ImageBufferMultiRoiDyn operator is used. The first images are simply stored beside the synchronization purposes are met by minimum frames being generated.

Using the VA simulation will help to understand how this works around SYNC in detail. In general a sequence of artificial minimum frames do the synchronization.

### 12.12.1.3. Visualization

Two different ways of visualization are implemented. The first will generate a 24 bit RGB image where :

- Plane - Content
- RED - current image
- GREEN - average image
- BLUE - difference image

The second approach is based on a HSL color space that in later converted into 24bit RGB :

- Plane - Content
- HUE / color angle - difference image, where minimum is green
- SATURATION - set to maximum
- LIGHTNESS - current image

The second approach is switched on by EffectEnable = 1. A dynamic gain factor can be used for difference scaling to all Hue color angles.

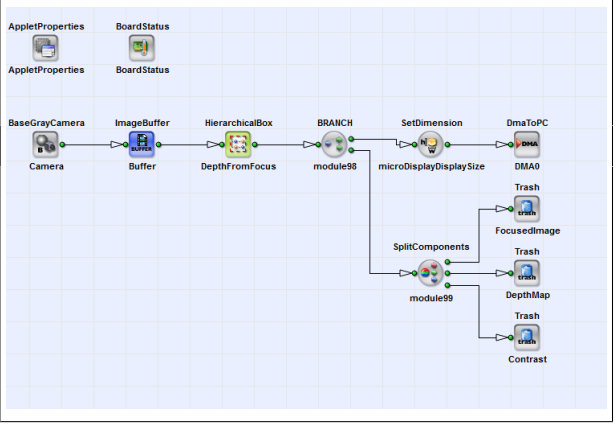
### 12.12.1.4. VisualApplets Design

The example "rollingAverage_Loop.va" is designed for a monochrome CameraLink camera in base configuration with a resolution of 1024x1024 at 8bit per pixel.

### 12.12.1.5. Simulation Data

The example "rollingAverage_Loop.va" folder %VASINSTALLDIR%\examples\Processing\Advanced \RollingAverage\street_sequence*.jpg includes a useful series of images for simulation. A lossy compression format was used in order to reduce amount of data.

## 12.12.2. Depth From Focus Using Loops

| **Brief Description** | |
|---|---|
| **Files:** \examples\Processing\Loop \DepthFromFocus\DepthFromFocus.va \examples\Processing\Loop\DepthFromFocus \DepthFromFocus_NoiseReduction.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Calculation of a focused image, a index depth map and the image contrast from an image sequence using loops. | |

The VisualApplets design "DepthFromFocus.va" calculates a completely focused image from a sequence of partially defocused images. The design "DepthFromFocus_NoiseReduction.va" gives in addition the possibility to reduce noise in the focused image. For every image of the sequence the focal setting of the camera, the image plane position or the object position (in axial direction) is changed. When doing so, a part of the object observed should always be in focus, whereas all other parts of the image are defocused. In addition in this design the highest local contrast of the image sequence is calculated with the corresponding image index map. This map gives information on the image from which the current pixel in the focused output image is taken. This provides the possibility to calculate the shape of the object observed. The focused image, the image index map and the contrast are sent to PC as red, green and blue color values. The design is implemented for a grayscale camera in CameraLink Base configuration for a marathonVCL board.

### 12.12.2.1. Theoretical Background

To calculate a completely focused image out of a sequence of partially defocused images, the local contrast for every single image is calculated. This contrast is determined with a high-pass filter:

$$h_p = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -24 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{12.46}$$

Is the local contrast in the current image higher than the local contrast in the previous image at this position, the pixel value from the current image is used. Otherwise the pixel value from the previous image is used. This procedure is repeated for the complete sequence of images. For every pixel in the completely focused image the index of the corresponding image with the highest local contrast is saved. This information gives the user the possibility (e.g for a thin lens, if the focal length and the image distance is known) to restore the shape of the object observed.

### 12.12.2.2. Implementation in Visual Applets

In Fig. 12.134 the basic design structure is shown. For a sequence of partially defocused images from a grayscale camera a completely focused image, the local contrast and an index map of the images with the highest local contrast are sent as RGB color components via **DmaToPC** to PC. The calculation is performed in the HierarchicalBox **DepthFromFocus**.

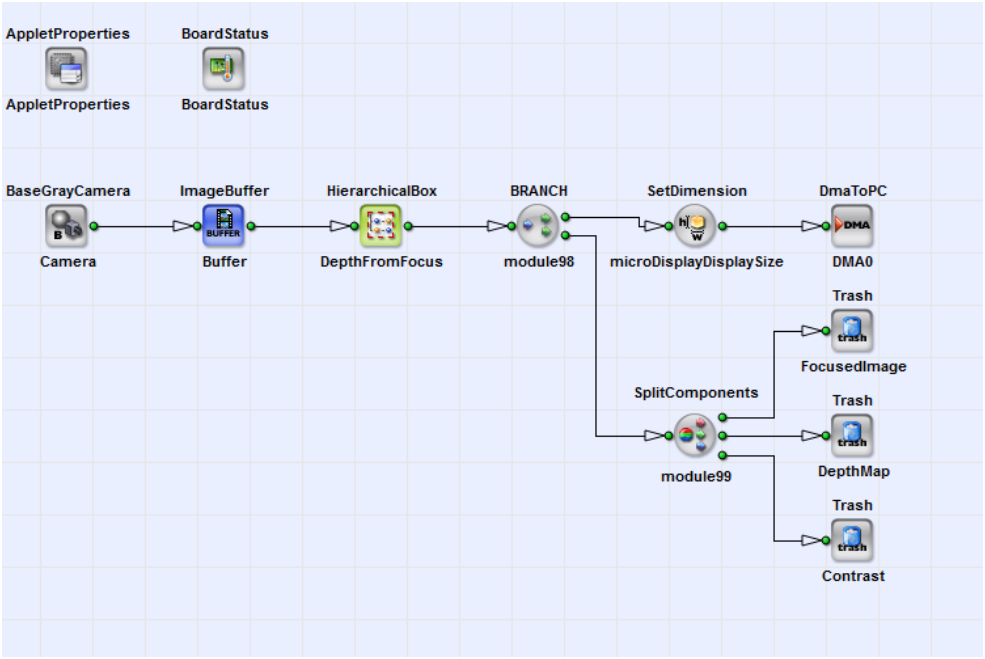



Figure 12.134. Basic design structure

In Fig. 12.135 you can see the content of the HierarchicalBox **DepthfromFocus**. With the transport parameters **IntParamTranslator_TranslateSequenceLength**, **IntParamReference_ReferenceImageWidth** and **IntParamReference_ReferenceImageHeight** you have the possibility to set the image sequence length, the image width and height in all operators necessary in the box **DepthFromFocus**. For this just perform a right mouse click on the box **DepthFromFocus**, select "properties" and set the corresponding parameters.

In the HierarchicalBox **HighpassFilter** the local contrast is calculated using a filter on a $5 \times 5$ kernel according to Equation 12.46. The upper output link of this box is the current pixel value. The lower output link of this box is the current local contrast. The operators **RxImageLink_Rx2_LoopContrast**/ **TxImageLink_Tx2_LoopContrast** represent the beginning and end of the loop calculating the highest local contrast from the sequence of images. As initial condition for the loop when starting the calculation on an image sequence, a blank image is inserted for synchronization and calculation reasons in the HierarchicalBox **InitAndResetContrast**.

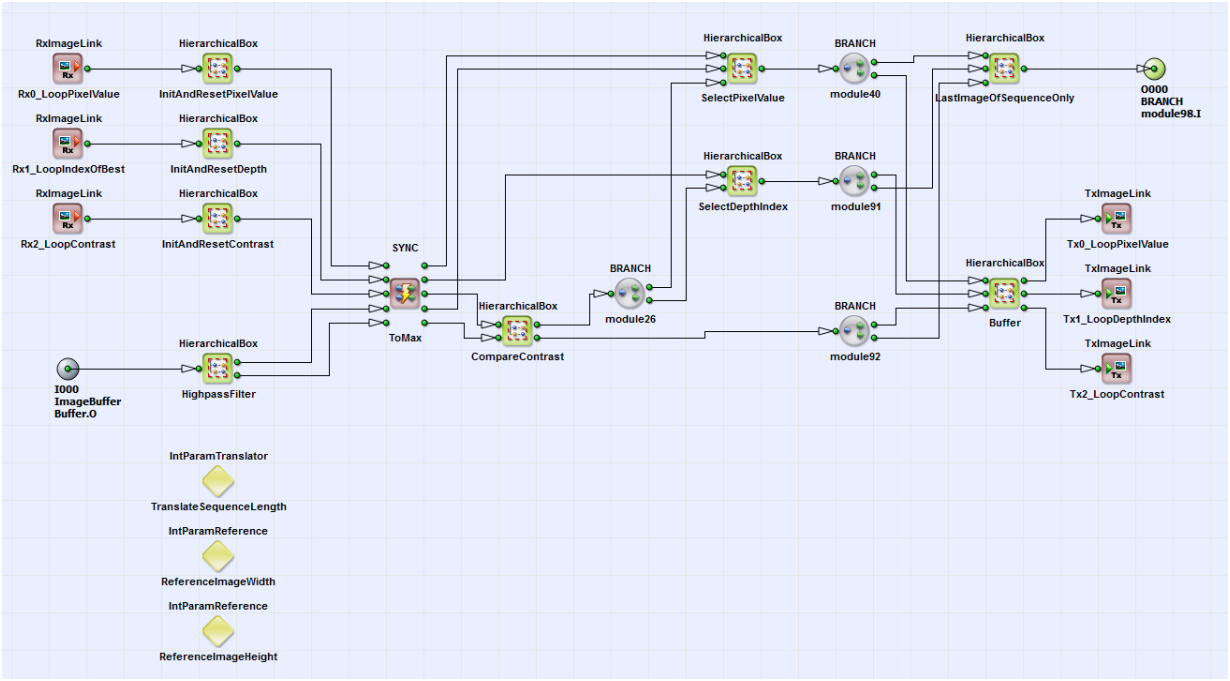



Figure 12.135. Content of HierarchicalBox **DepthFromFocus**

In the HierarchicalBox **CompareContrast** the current local contrast is compared to the local contrast at this position in the previous image of the sequence. Fig. 12.136 shows the content of this box. If the current local contrast is higher than the one in the previous image, the current local contrast is forwarded to the lower output link of this HierarchicalBox. In the opposite case the value of the previous local contrast is forwarded. The upper output link of the box **CompareContrast** is a index 0 (current contrast is stronger) or 1 (previous contrast is stronger).

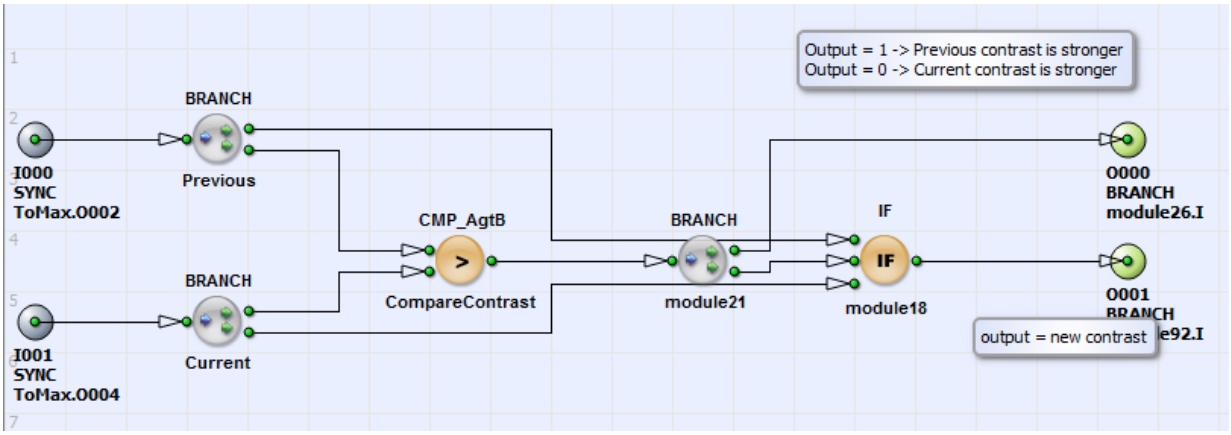



Figure 12.136. Content of HierarchicalBox **CompareContrast**

In the HierarchicalBox **SelectDepthIndex** (content see Fig. 12.137) in the loop **RxImageLink_Rx1_LoopIndexOfBest/TxImageLink_Tx1_LoopDepthIndex** the current image index (operator **ModuloCount_SequenceIndexCount**) is selected, if the current image has the highest local contrast. Otherwise the image index of the previous image is forwarded to the output link of this box. This procedure is repeated for the complete image sequence. As result the image indices with the highest local contrasts of the sequence are calculated. The initial condition with an insert of a blank image is performed in the box **InitAndResetDepth**.

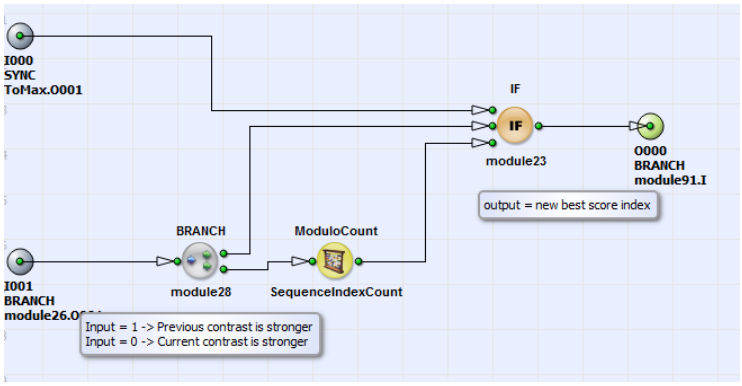



Figure 12.137. Content of HierarchicalBox **SelectDepthIndex**

The completely focused image out of the image sequence is calculated in the loop **RxImageLink_Rx0_LoopPixelValue**/ **TxImageLink_Tx0_LoopPixelValue** with the calculation performed in the box **SelectPixelValue**. In Fig. 12.138 its content is shown. The current pixel value is forwarded to the output of the **IF** operator if the current local contrast is higher than the local contrast in the previous image. This procedure is repeated for the complete image sequence. At the end of the sequence a completely focused image is forwarded to the input of the box **LastImageOfSequenceOnly**.

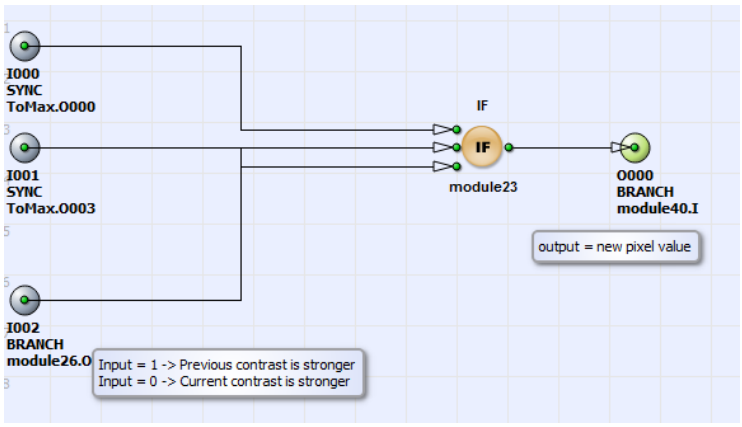



Figure 12.138. Content of HierarchicalBox **SelectPixelValue**

In Fig. 12.139 the content of this box is shown. The pixel values with the currently highest local contrasts, the corresponding image depth index and the currently highest local contrast are combined as RGB color components. Only the last RGB image with the completely focused image, the highest local contrast and the corresponding image depth index is forwarded to the output of the box and via DMA transport to PC.

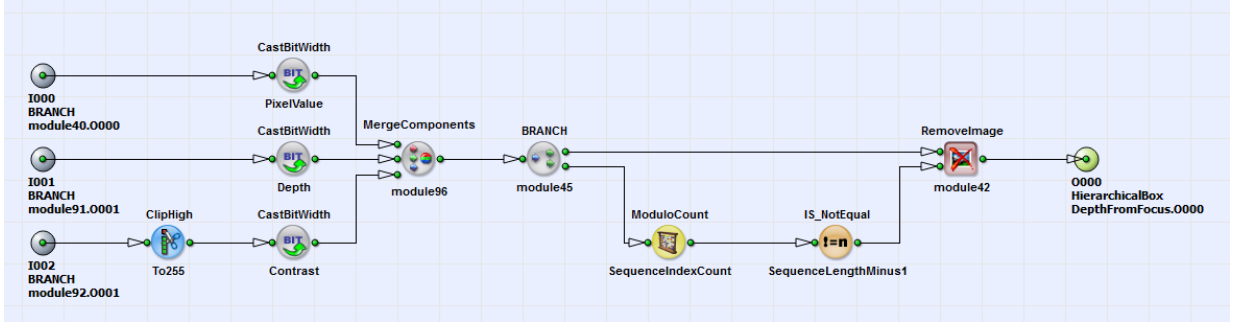


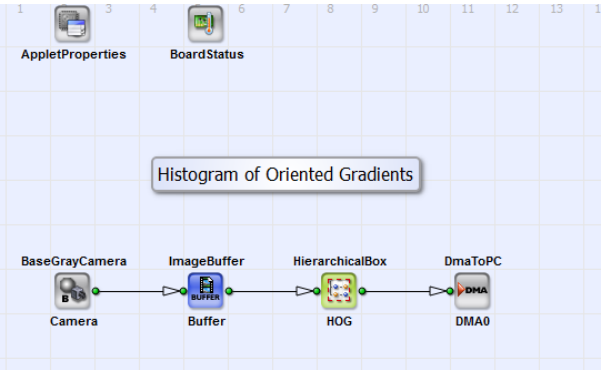

Figure 12.139. Content of HierarchicalBox **LastImageOfSequenceOnly**

In the VA design "DepthFromFocus_NoiseReduction" there is the possibility to reduce with an adaptive Median filter the local noise in the resulting focused image. You find this filter in the HierarchicalBox **AdaptiveMedian**. If you have a local noise at the current pixel the local Median is selected. If there is no local noise at the current position the current pixel value is selected. A local noise is determined with a lowpass filter and an adaptive threshold **IS_GreaterThan_Threshold**. With this method it can be prevented that contrasts of local structures are lost but noise is reduced.

## 12.13. Object Features

The examples in the following perform calculation of object features such as Histogram of Oriented Gradients (HOG). Also a print inspection example is provided in this section. In this design an object is position and orientation corrected and defects are detected.

### 12.13.1. Histogram of Oriented Gradients (HOG)

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Geometry\ObjectFeatures\HOG\HOG_4Bins_HistogramMax.va \examples\Processing\Geometry\ObjectFeatures\HOG\HOG_9Bins_HistogramMax.va \examples\Processing\Geometry\ObjectFeatures\HOG\HOG_9Bins_Histogram.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Calculation of Histogram of Oriented Gradients (HOG) | |

The VisualApplets design examples "HOG_4Bins_HistogramMax.va", "HOG_9Bins_HistogramMax.va" and "HOG_9Bins_Histogram.va"calculate the Histogram of Oriented Gradients (HOG) for a grayscale image with maximum dimensions of 1024x1024 pixels.The algorithm is mainly based on the publication of [Dal05]. The HOG is a feature descriptor and can be used for object recognition. In the following subsections first the theory is described before the applet design is introduced.

#### 12.13.1.1. Theory

The image gradient orientation and magnitude is calculated for each pixel in the image. The gradient filter kernels $s_x, s_y$ in x- and y-direction are:

$$s_x = [-1 \quad 0 \quad 1] \text{ and}$$
$$s_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}. \tag{12.47}$$

The gradient magnitude $m_G$ and orientation $\theta_G$ is then calculated as:

$$m_G = \sqrt{s_x^2 + s_y^2},$$
$$\theta_G = \arctan\frac{s_y}{s_x}. \tag{12.48}$$

In dependence on their orientation the calculated gradients are then assigned to a certain number of orientation bins. The vote for the bin is a function of the gradient magnitude. As Dalal and Triggs found

out, a splitting of the orientation into 9 bins over $180^\circ$ give the best results when using the HOG features for human detection methods [Dal05]. In the VA example designs "HOG_9Bins_HistogramMax.va" and "HOG_9Bins_Histogram.va" we have implemented such binning. In addition in the example "HOG_4Bins_HistogramMax.va" we perform a splitting into 4 bins over $180^\circ$. The assignment or weighting $w_{Bin}$ of a certain orientation $\Theta_G$ to a bin (with orientation $\mathrm{X}_{Bin}$) is done via interpolation:

$$w_{Bin} = \frac{||\Theta_G - \mathrm{X}_{Bin}| - \Phi_{Bin}|}{\Phi} \text{ if } ||\Theta_G - \mathrm{X}_{Bin}| - \Phi_{Bin}| < \Phi \\ \text{else } w_{Bin} = 0. \tag{12.49}$$

Here $\Phi_{Bin}$ is the size of angle steps between the bins. A single bin column $c_H$ of the Histogram of Oriented Gradients (each for a region of 8x8 pixels) is then calculated as:

$$c_H = \sum_{i=1}^{64} w_{Bin} \cdot m_G. \tag{12.50}$$

The histograms for each cell of 8x8 pixels are then grouped in blocks of 2x2 cells size to cover local variances in luminance or contrast. The blocks have an overlap of $50\%$. Using the HOG descriptor for object recognition purposes, block normalization (algorithm see [Dal05]) improves the performance by a factor of $27\ \%$. Block normalization is omitted in the current design examples. In the designs "HOG_4Bins_HistogramMax.va" and "HOG_9Bins_HistogramMax.va", the maximum histogram orientation is forwarded to DMA, whereas in "HOG_9Bins_Histogram.va" the complete Histogram of Oriented Gradients is sent to PC. The designs introduced in the following can easily be adapted to the special purpose of the user. That is, a certain amount and step size of the orientation binning or a block normalization according to [Dal05] can be implemented in addition.

### 12.13.1.2. Implementation in VisualApplets

In Fig. 12.140 you can see the basic design structure of the VA design for the calculation of the HOG feature. For a grayscale image from a camera in CameraLink base configuration the Histogram of Oriented Gradients is calculated in the HierarchicalBox **HOG**. This histogram is sent to PC via DMA in the design "HOG_9Bins_Histogram.va". In the VA designs "HOG_4Bins_HistogramMax.va" and "HOG_9Bins_HistogramMax.va" the maximum component is forwarded to PC. In the following we explain the design structure of "HOG_4Bins_HistogramMax.va". It is analog for the other two designs. We explain possible differences in the designs for the corresponding issues. The default platform is marathon VCL. The design implementations can easily adapted to other platforms or camera configurations.

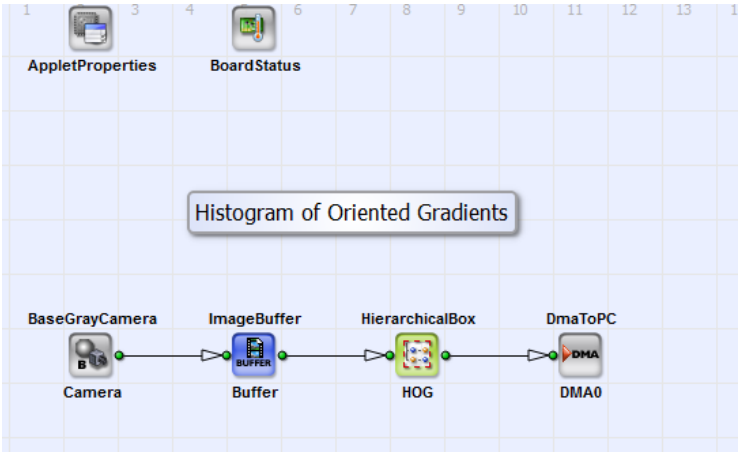



Figure 12.140. Basic Design structure of the VA designs "HOG_9Bins_Histogram.va","HOG_9Bins_HistogramMax.va" and "HOG_4Bins_HistogramMax.va"

You can see the content of the HierarchicalBox **HOG** in Fig. 12.141.

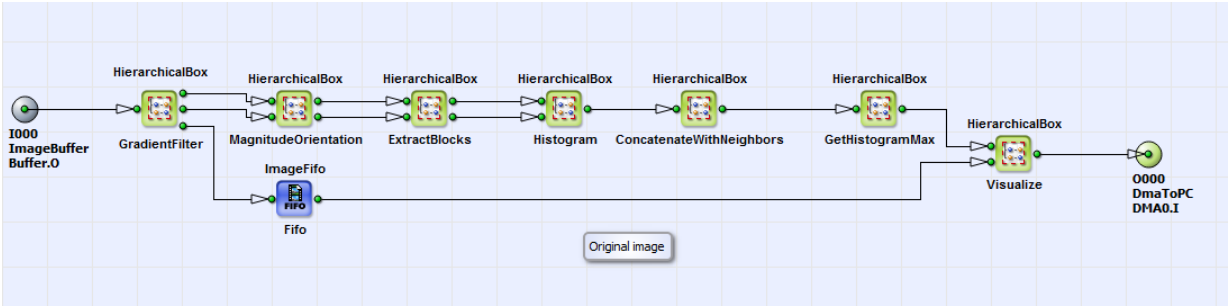



Figure 12.141. Content of HierarchicalBiox **HOG**

For the original input image the gradients in x- and y- direction are calculated in the HierarchicalBox **GradientFilter**. The implementation is shown in Fig. 12.47. The operator **FIRkernelNxM** generates a 3x3 kernel. The operators **FIRoperatorNxM_VerticalEdges** and **FIRoperatorNxM_HorizontalEdges** select according to eq. 12.47 the gradients in x- and y-direction.

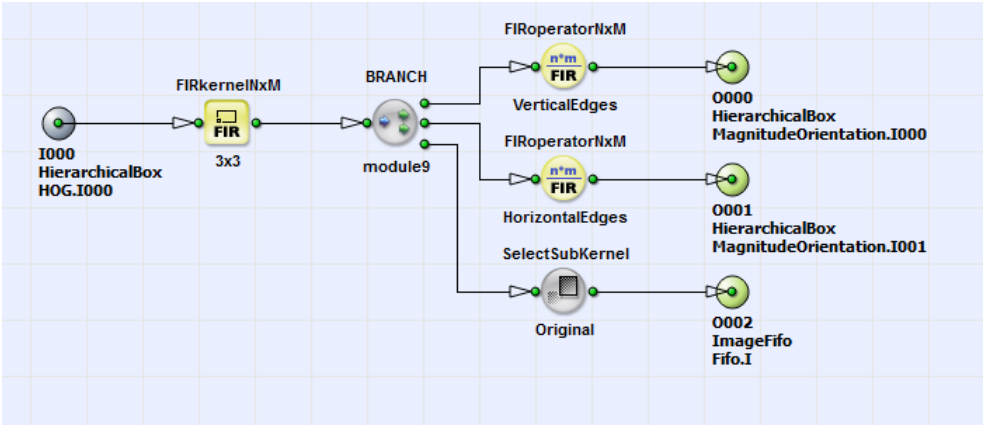



Figure 12.142. Content of HierarchicalBox **GradientFilter**

For this gradients the magnitude and orientation according to eq. 12.48 is implemented in the box **MagnitudeOrientation** (content and structure see Fig. 12.143).Detailed comments in the box help to understand the implemented formula.

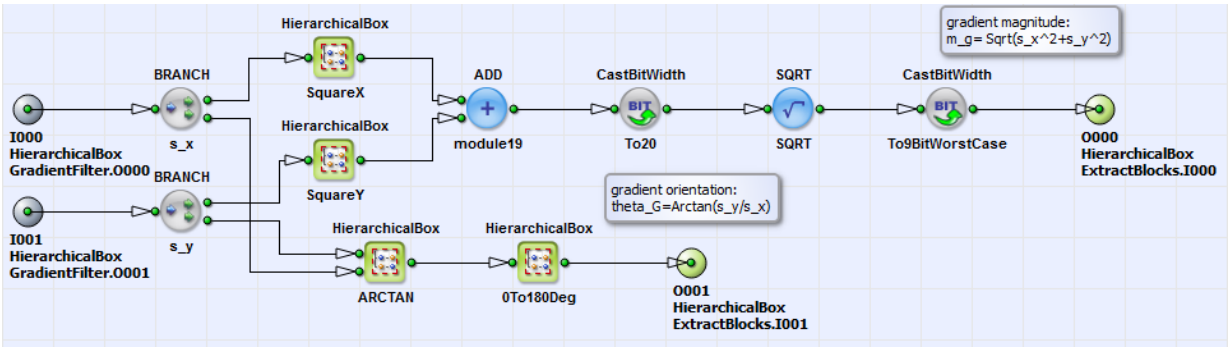



Figure 12.143. Content of HierarchicalBox **MagnitudeOrientation**

The grouping of the pixels in cells of 8x8 pixels is performed in the HierarchicalBox **ExtractBlocks**. The output links of this box forward the gradient magnitude and orientation of the cells in form of lines of size 64 pixels. This information is input for the calculation of the Histogram of Oriented Gradients, which is performed in the HierarchicalBox **Histogram**. You can see its content in Fig. 12.144.

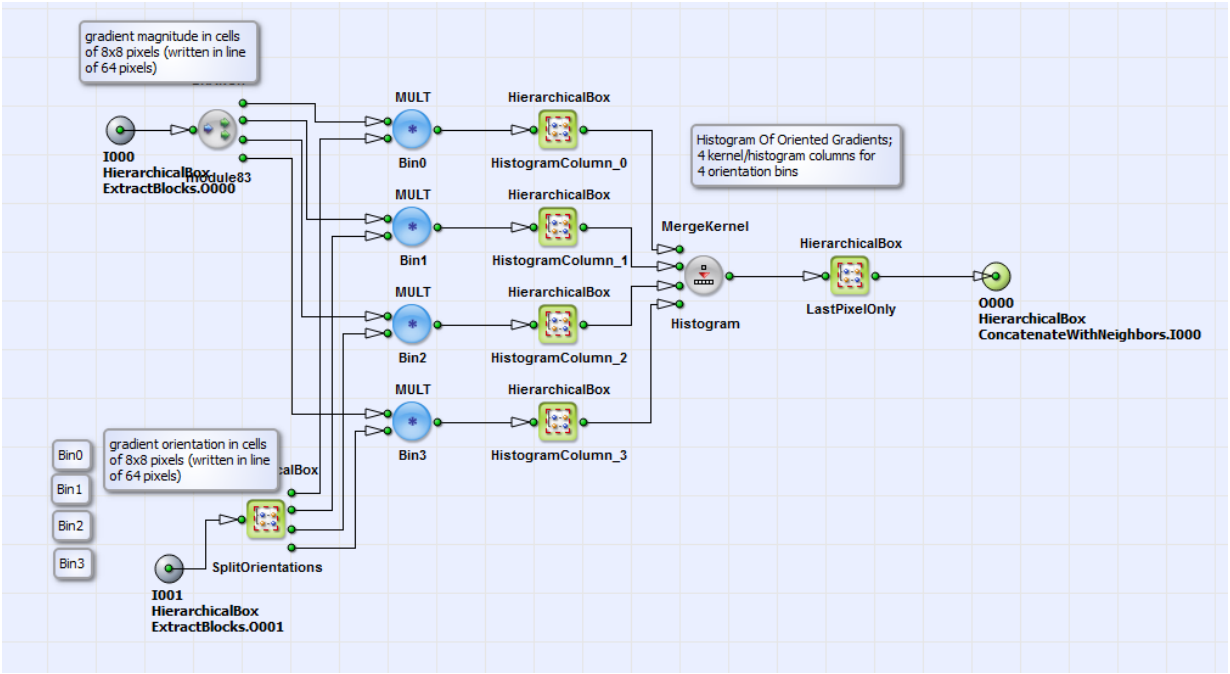



Figure 12.144. Content of HierarchicalBox **Histogram**

Here an assignment of the gradient orientations, which can be in a region of $0°$ to $180°$, to four bins is performed in the box **SplitOrientations**. The bins are $0°$, $45°$, $90°$ and $135°$. For the designs "HOG_9Bins_Histogram.va" and "HOG_9Bins_HistogramMax.va" nine bins ( $0°$ to $160°$ in steps of $20°$) exist. The assignment of the gradient orientation to a certain bin is done the way, that a weight of this orientation for the bin is calculated. As an example the weighting and therefor assignment to bin 1, which is $45°$, is shown in Fig. 12.145.

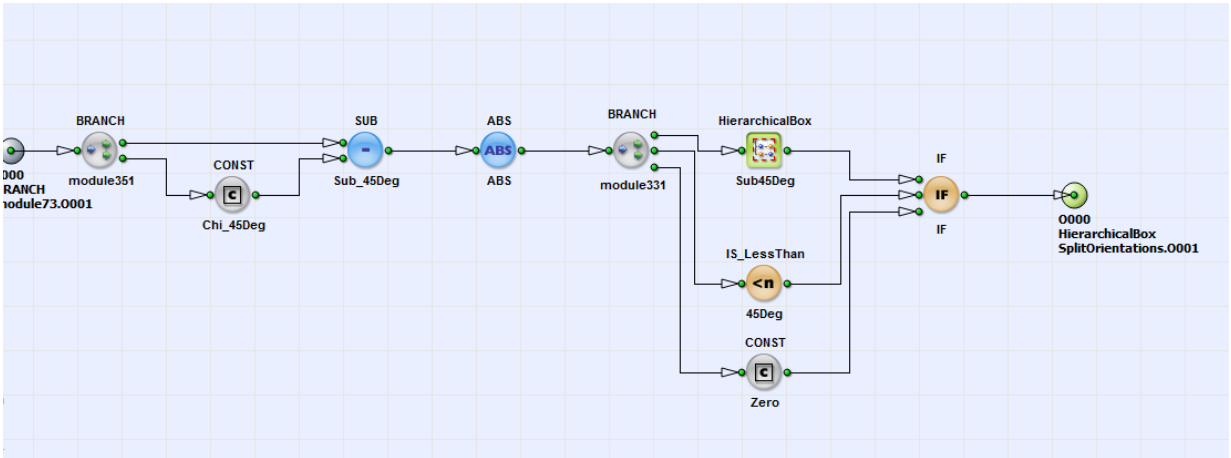



Figure 12.145. Content of HierarchicalBox **Bin1**

Here the weight $w_{Bin1}$ according to eq. 12.49 is calculated for bin 1. The angle distance $\Phi_{Bin}$ between the bins is in this example $45°$ (operator **CONST_Phi_45Deg** in box **Sub45Deg**). The angle bin orientation for bin 1 $X_{Bin}$ is also $45°$ (operator **CONST_Chi_45Deg**). The weighting factor for each bin is then multiplied with the gradient magnitudes (see Fig. 12.144). In the boxes **HistogramColumn_0** to **HistogramColumn_3** the content of each histogram column (for each bin) according to eq. 12.50 is calculated as the sum of the weighted gradient magnitudes over all image pixels. The Histogram of Oriented Gradients is then generated with merging the histogram columns with the operator **MergeKernel_Histogram**. Only the last pixel of each kernel component is the true value of the histogram columns, so it is selected in the box **LastPixelOnly**. To account for local gradient changes the Histogram of Oriented Gradients for each cell is combined to blocks of size 2x2 cells per block with

an overlap of $50\ \%$ (see theory part of this section). The combination of the cells to blocks is performed in the box **ConcatenateWithNeighbors**. See its content in Fig. 12.146.

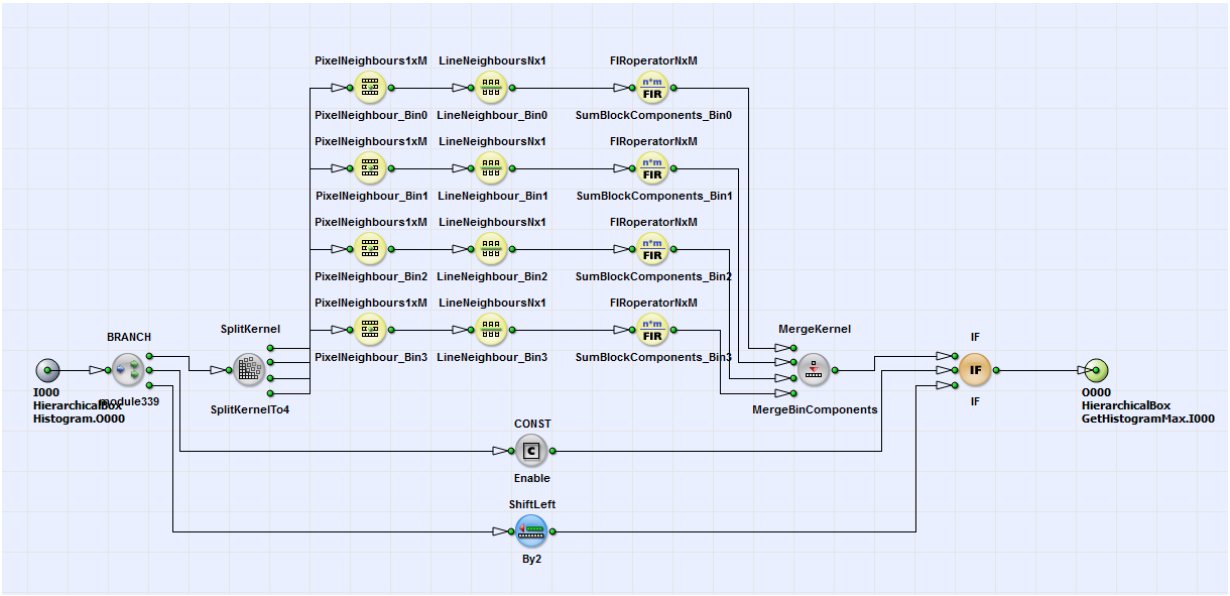



Figure 12.146. Content of HierarchicalBox **ConcatenateWithNeighbors**

For each cell (represented by a single pixel) with information about the Histogram of Oriented Gradients, the histogram columns for each bin are separated with the operator **SplitKernel**. For each kernel component the next cell neighbors in x and y- direction are calculated with the operators **PixelNeighbours1xM** and **LineNeighboursNx1**.The operator **FIRoperatorNxM** sums the information of the 2x2 cells up. No normalization is performed here. As an alternative implementation the grouping of the cells to blocks can be omitted by setting the operator **Const_Enable** to zero. The maximum of the calculated Histogram of Oriented Gradients is selected in box **GetHistogramMax** (see Fig. 12.141). Its content is shown in Fig. 12.147. In the design "HOG_9Bins_Histogram.va" the selection of the histogram maximum is omitted.

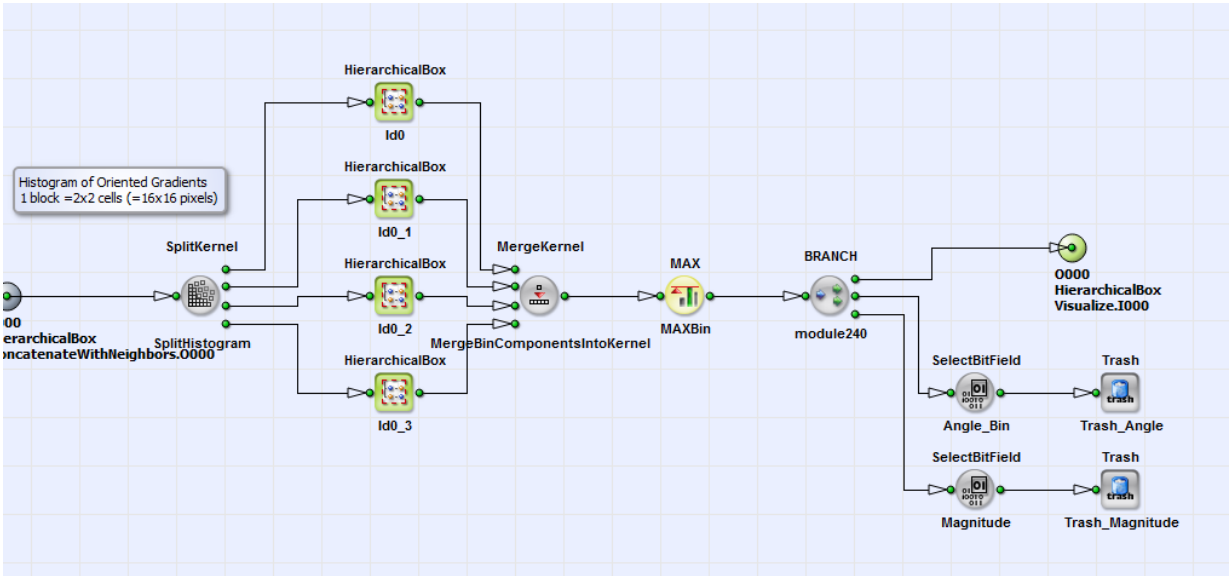
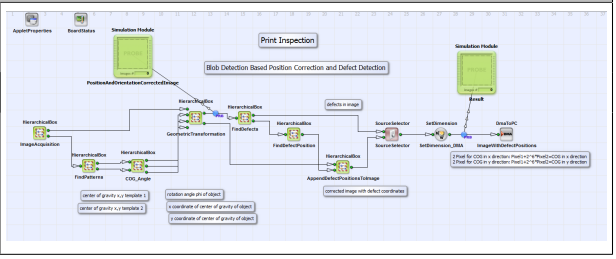



Figure 12.147. Content of HierarchicalBox **GetHistogramMax**

Finally the maximum of the histogram or the histogram itself is visualized in box **Visualize** (see Fig 12.141) as red lines. In this box (path: box **Draw**, **EnableOriginal**) you have the possibility to visualize

the original image for a number of frames together with the HOG feature by setting the parameters **ModuloCount_Period** (counts frames), **Is_InRange** (set frame range) and **Const_Enable** or always (setting **Const_Enable** to one). As last step the HOG feature is transferred to PC via the operator **DmaToPC_DMA0**. Here the HOG feature can be used as descriptor for object detection in Support Vector Machines (SVP) or also Convolutional Neural Networks (CNN).

## 12.13.2. Print Inspection Example- Position Correction and Defect Detection Using Blob Based Template Matching

| **Brief Description** | |
|---|---|
| **Files:** \examples\Processing\ObjectFeatures \PrintInspection\PrintInspection_Blob.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Blob based position correction and defect detection | |

A position and orientation correction of an object and subsequent defect detection in an image is performed in the design "PrintInspection_Blob.va". The example is suitable for objects with at least two imprints, which maybe used as templates. A special imprint should only occur once on the object. In the example image "TestImage.tif" the letters "K" and "g" are selected as template 1 and 2 with a size of 29x38 pixels. In the comment box on the top layer of the design the single steps to adapt the design to your example image are described. In the design the position and orientation of an object is determined with the blob detection based recognition of two templates on the object. From the coordinates of the two templates the center of gravity of the object is calculated. From the relative position of template 1 and 2 the rotation angle Phi of the object is calculated. With these informations a geometric transformation as described under 12.9.3 is performed. The implementation of this transformation is equivalent to the design example "Geometric Transformation_ImageMoments.va" described under 12.9.3.2.2. Defects on the position and orientation corrected object are found in subtracting a "golden master" template of the object from the calculated corrected object. What remains are possible defects. Via blob detection the positions of these defects are determined and attached to the image of the position and orientation corrected object. The user can choose whether he/she wants to transfer the image with defects or the image with the position corrected object and the attached defect positions to PC. In the following the basic design structure and the functionality of the single modules are explained. In Fig. 12.148 you can see the basic design structure.

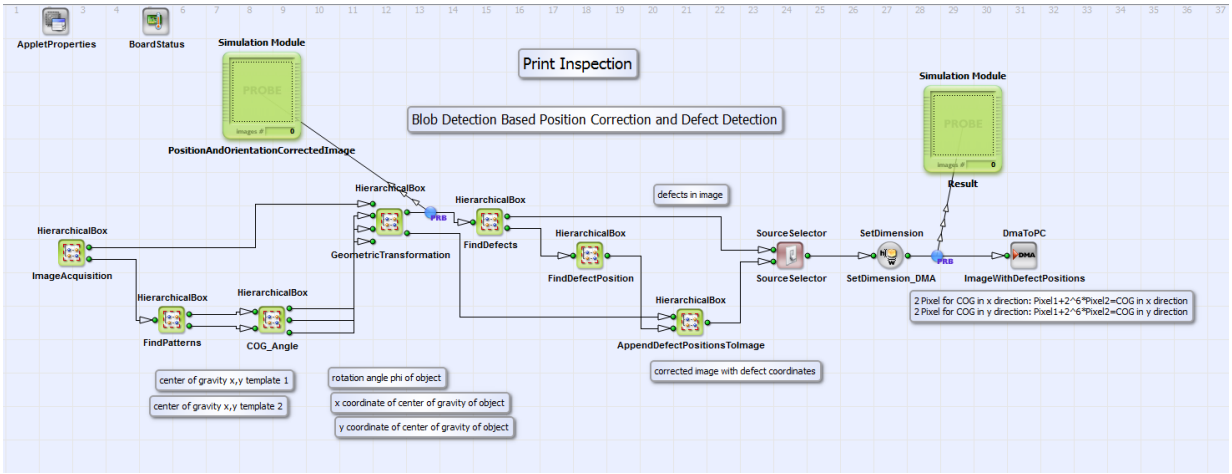



Figure 12.148. Basic design structure of the VA design "PrintInspection_Blob.va"

In the HierarchicalBox **FindPatterns** (see Fig. 12.149) the acquired image (from box **ImageAcquisition**) is binarized in the HierarchicalBox **Binarization** with a simple threshold. When you change the example image you may adjust this threshold to your image. In the HierarchicalBox

**BlobDetection** objects in the binarized image are detected with the operator **Blob_Analysis_2D**. Output of this operator are the x and y coordinates and the area of a bounding box around each object found. In the comment box in the design you find more detailed information on these parameters. The objects found are then selected in the box **BlobSelection** with respect to their size and position. Only the coordinate and size informations for the template relevant objects remain. Please adjust the parameters **IS_InRange** for the valid size and position of the objects to your templates (right-mouse-click on box **FindPatterns**). The coordinate and size informations are input for the HierarchicalBox **ExtractCandidates**. See its content in Fig. 12.150.

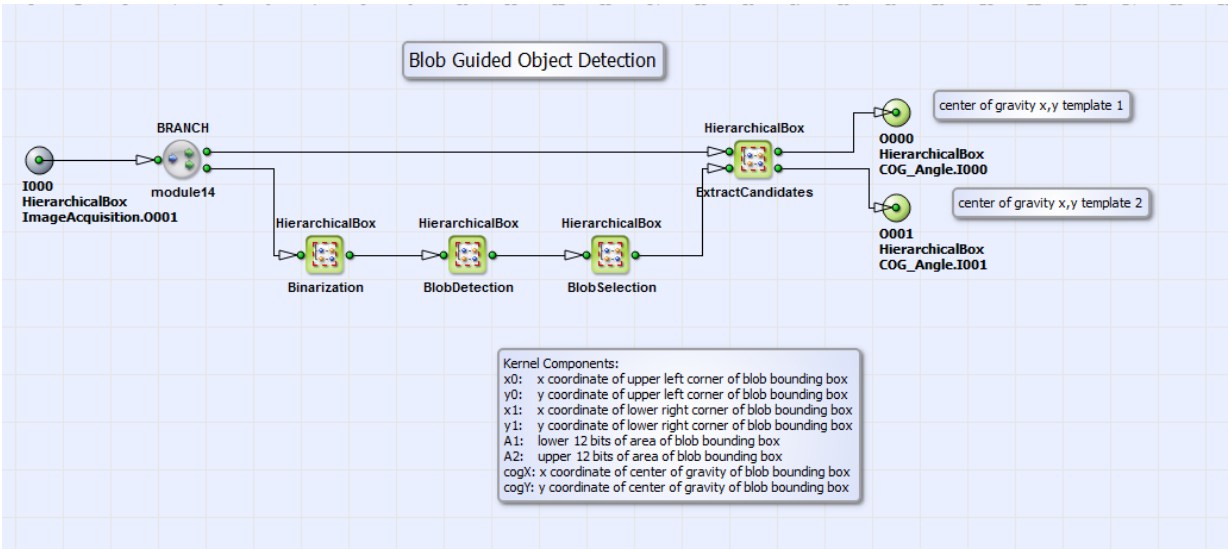



Figure 12.149. Content of the HierarchicalBox **FindPatterns**

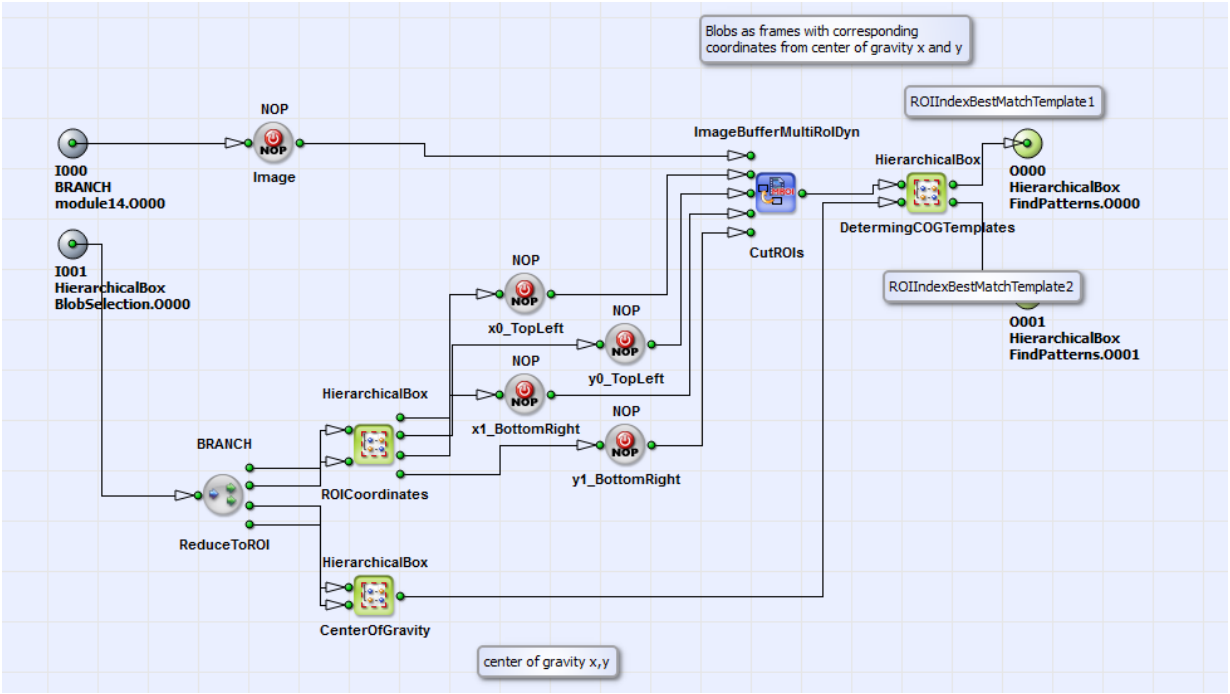



Figure 12.150. Content of the HierarchicalBox **ExtractCandidates**

Here each object (detected with blob analysis) is cut from the original image as region of interest with the operator **ImageBufferMultiROIDyn_CutROIs** and is forwarded as single frame. In the HierarchicalBox **DeterminingCOGTemplates** (see Fig. 12.151) the template matching and the subsequent determination of their coordinates in the image is performed.

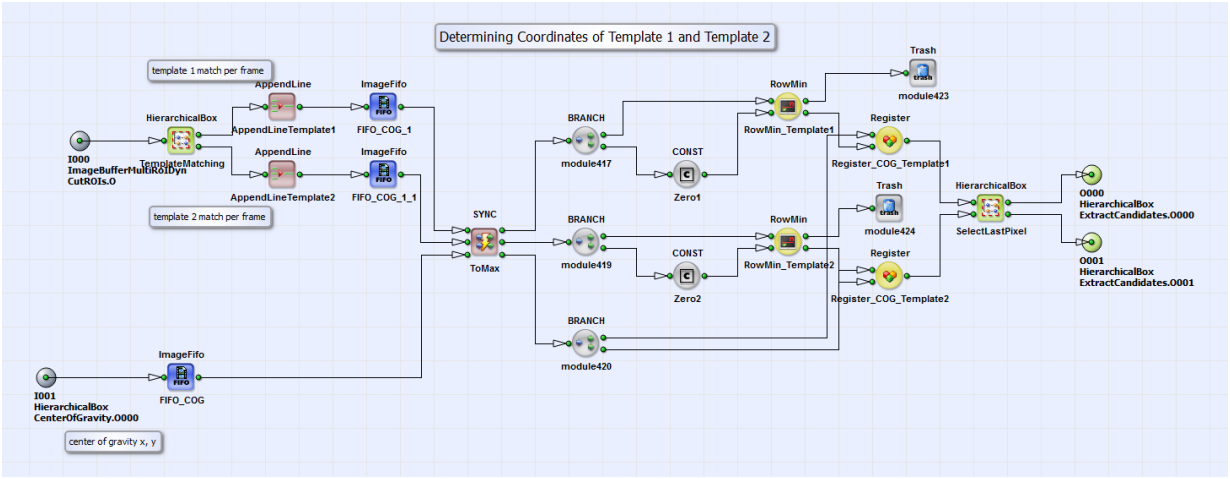



Figure 12.151. Content of the HierarchicalBox **DetermingCOGTemplates**

In the box **TemplateMatching** (see its content in Fig. 12.152) the matching of the selected objects with template 1 and 2 (letters "K" and "g" with a size of 29x38 pixels) is performed. In **Templates** template 1 and 2 are written as 16 bit values (8 bit for each template) to the lookup table **LUT_Values**. The matching of the objects found in the image with template 1 and 2 is the performed as simple 3x3 kernel subtraction in the box **Match**. In **TemplateMatch** the kernel component which gives the best match is selected. Going back to the content of **DeterminingCOGTemplates**, the best template match is selected with the operators **RowMin_Template1** and **RowMin_Template2** and the corresponding coordinates of template 1 and 2 with the registers **Register_COG_Template1**, **Register_COG_Template2** and the selection of the last pixel in box **SelectLastPixel**. The output of the box **DeterminingCOGTemplates** are the coordinates of template 1 and 2.

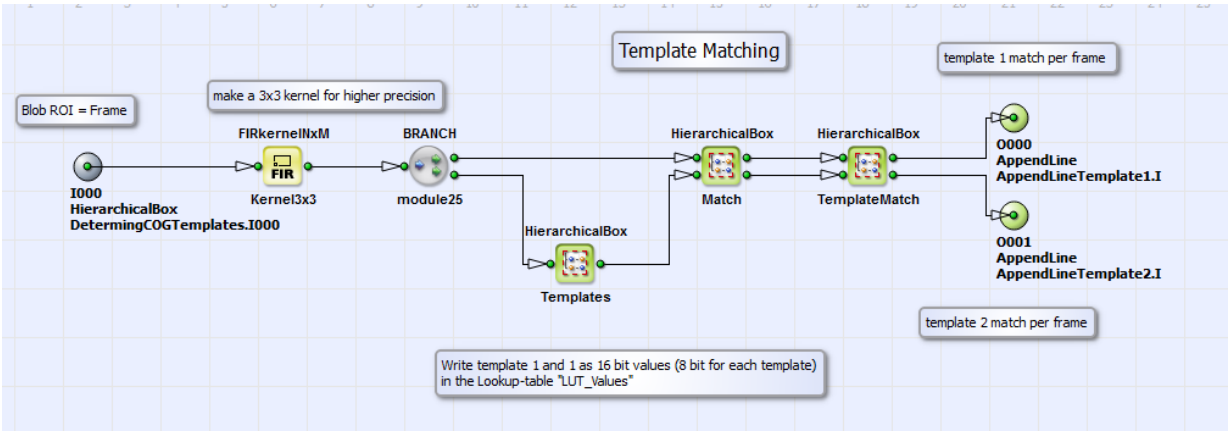



Figure 12.152. Content of the HierarchicalBox **TemplateMatching**

Going back to the basic design structure we will now have a look in the HierarchicalBox **COG_Angle**, in which the coordinates of the center of gravity of the object and its rotation angle is determined from the coordinates of template 1 and 2. See the content of **COG_Angle** in Fig. 12.153.

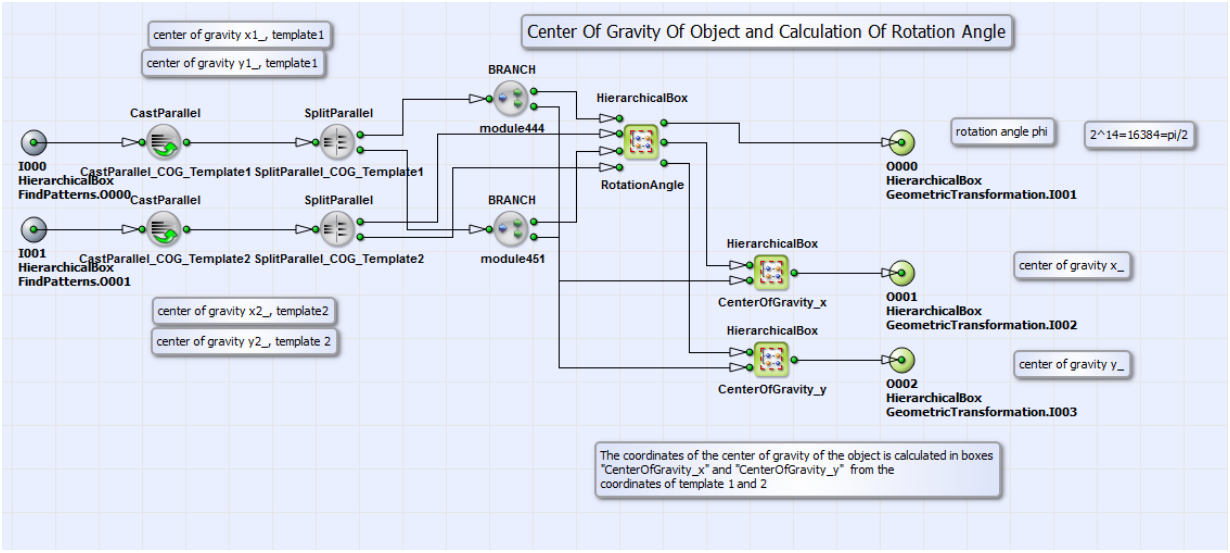



Figure 12.153. Content of the HierarchicalBox **COG_Angle**

The rotation angle $\Phi$ of the object is calculated in box **RotationAngle** according to

$$\Phi = \arctan(\frac{dy}{dx}) \ , \tag{12.51}$$

where $dx$ and $dy$ are the differences of the coordinates of template 1 and 2 in x and y direction.

The coordinates of the center of gravity of the object $x_{COG}$ and $y_{COG}$ are calculated according to to

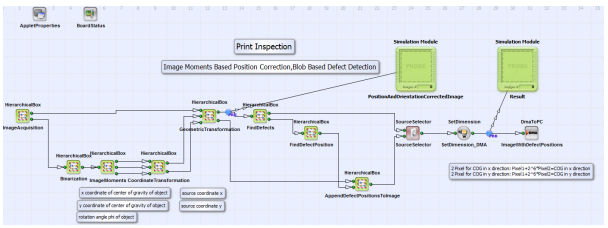
$$\begin{aligned} x_{COG} &= \cos(\Phi + \alpha + \mathrm{X}) \cdot z \ , \\ y_{COG} &= \sin(\Phi + \alpha + \mathrm{X}) \cdot z \ . \end{aligned} \tag{12.52}$$

Here $\alpha$ is the angle between the center of gravity of the object, template 1 and template2, $\mathrm{X}$ is the angle between template2, template 1 and a horizontal line, when the object is not rotated and $z$ is the distance between the center of gravity of the object and template 1. When you change the example image you have to adapt these values with right-mouse-click on the box **COG_Angle** under "properties".

With these information about $\Phi$, $x_{COG}$ and $y_{COG}$ the geometric transformation is performed in box **GeometricTransformation"** (see basic design structure in Fig. 12.148) equivalent to the VA example "GeometricTransformation_ImageMoments.va" described in section 12.9.3.2.2.

Possible defects in the position and orientation corrected image as output of box **GeometricTransformation** are found in box **FindDefects**, where a "golden master" template of the image is subtracted from the processed image. After elimination of remains (a result of bilinear interpolation after geometric transformation) the output of the box **FindDefects** are the pure defects found on the object. Via blob analysis in box **FindDefectPosition** with the operator **Blob_Analysis_2D** the coordinates of these defects are determined. In the box **AppendDefectPositionsToImage** these coordinates are appended to the position and orientation corrected image. The user can choose whether he/she wants to send the image with pure defects or the corrected image with defect coordinates via DMA to PC in setting the parameter "SelectSource" of the operator **SourceSelector** to 0 or 1.

## 12.13.3. Print Inspection Example- Position Correction and Defect Detection Using Image Moments and Blob Based Template Matching

| Brief Description | |
|---|---|
| **Files:** \examples\Processing \ObjectFeatures\PrintInspection \PrintInspection_ImageMoments.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Image moments based position correction and defect detection via blob detection | |

A position and orientation correction of an object and subsequent defect detection is performed in the design "PrintInspection_ImageMoments.va". The example is suitable for objects with no imprints. You can see the basic design structure in Fig. 12.154. The acquired image object is binarized in the HierarchicalBox **Binarization** and via image moments (see theory in section 12.9.5) the orientation and position of the object is determined (content of box **ImageMoments**). The parameter for the binarization threshold is suitable for the example image "testImage.tif". If you use your example image please adapt the binarization threshold. A geometric transformation is performed to correct position and orientation equivalent to the example "GeometricTransformation_ImageMoments" (see section 12.9.3.2.2). The detection of defects and their coordinates (in boxes **FindDefects** and **FindDefectPosition**) is equivalent to the defect detection in the print inspection example "PrintInspection_Blob.va" (see section 12.13.2). The user can choose whether he/she wants to send an image with the pure defects or the combined image (position and orientation corrected image together with the defects coordinates as output from box **AppendDefectPositionToImage**) to PC in setting the parameter "SelectSource" of the operator **SourceSelector** to 0 or 1.

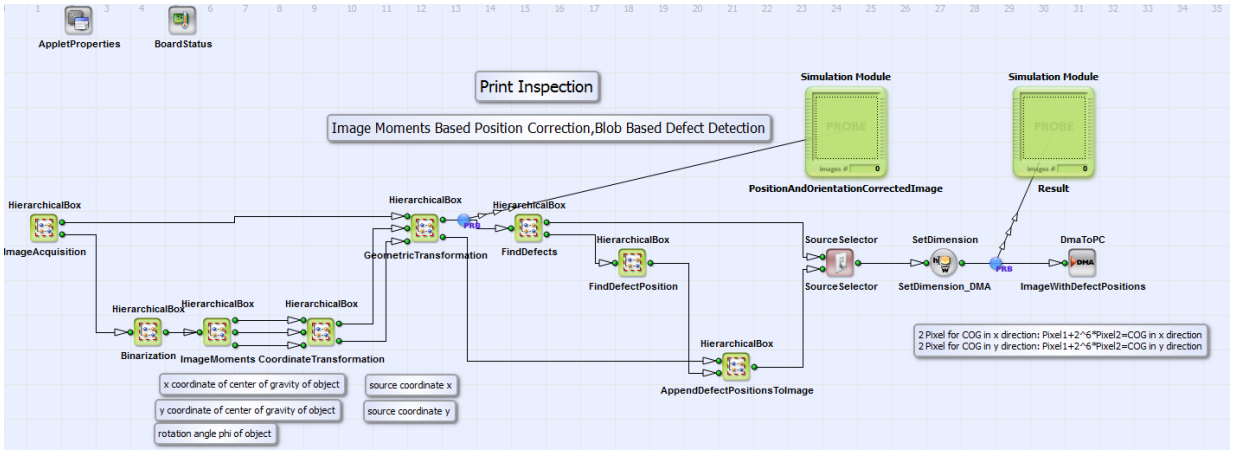
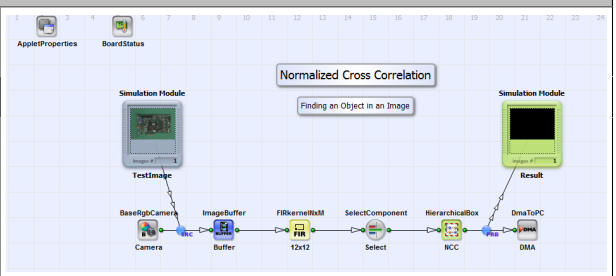



Figure 12.154. Basic design structure of the VA design "PrintInspection_ImageMoments.va"

## 12.13.4. Normalized Cross Correlation

| Brief Description | |
|---|---|
| **Files:** \examples\Processing\ObjectFeatures \NormalizedCrossCorrelation.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A reference object is identified in an image. The output image is a binary image with " 1" at object position. | |

### 12.13.4.1. Theory

Template matching based on normalized-cross-correlation (NCC) algorithm uses the minimization of squared Euclidean distance $d_E^2$ at position $x,\ y$

$$d_E^2(x,y) = \sum_{u,v}(I(x+u,y+v) - R(u,v))^2, \tag{12.53}$$

to find best matching between a reference object $R(u,v)$ (with dimension $uxv$) and a region $I(x+u,y+v)$ in an image at position $x,\ y$. The Euclidean distance is minimized, when the linear cross correlation coefficient $C_L(x,y)$

$$C_L(x,y) = \sum_{u,v} I(x+u,y+v) \cdot R(u,v) \tag{12.54}$$

between the reference object $R(u,v)$ and image region $I(x+u,y+v)$ is maximized. To account for intensity variations in the image and make the correlation coefficient invariant to pixel intensities, the correlation between the difference of object $R(u,v)$ to the mean value $\bar{R}(u,v)$ and image region $I(x+u,y+v)$ to the mean value $\bar{I}(x+u,y+v)$ is calculated. According to [Bur06] the so called normalized-cross correlation coefficient can then be expressed as:

$$C_L(x,y) = \frac{\sum_{u,v}\left(I(x+u,y+v) \cdot R(u,v)\right) - K \cdot \bar{I}(x,y) \cdot \bar{R}}{\sqrt{\sum_{u,v}\left(I(x+u,y+v)\right)^2 - K \cdot \left(\bar{I}(x,y)\right)^2} \cdot \sigma_R} \tag{12.55}$$

with

$$\sigma_R = \sqrt{\sum_{u,v}(R(u,v)^2 - K \cdot \bar{R}^2} \tag{12.56}$$

The result of $C_L(x,y)$ is between -1 and 1. The higher the accordance between reference image $R(u,v)$ and image region $I(x+u,y+v)$ is, the higher is result of $C_L(x,y)$. The position $x,\ y$ in image $I$ with highest value of $C_L(x,y)$, is the location where the reference object $R(u,v)$ is found.

### 12.13.4.2. Implementation in VisualApplets

In "NormalizedcrossCorreslation.va" an object is identified in an image using the NCC algorithm according to eq. 12.55. In Fig. 12.155 you can see the basic design structure of "NormalizedCrossCorrelation.va".

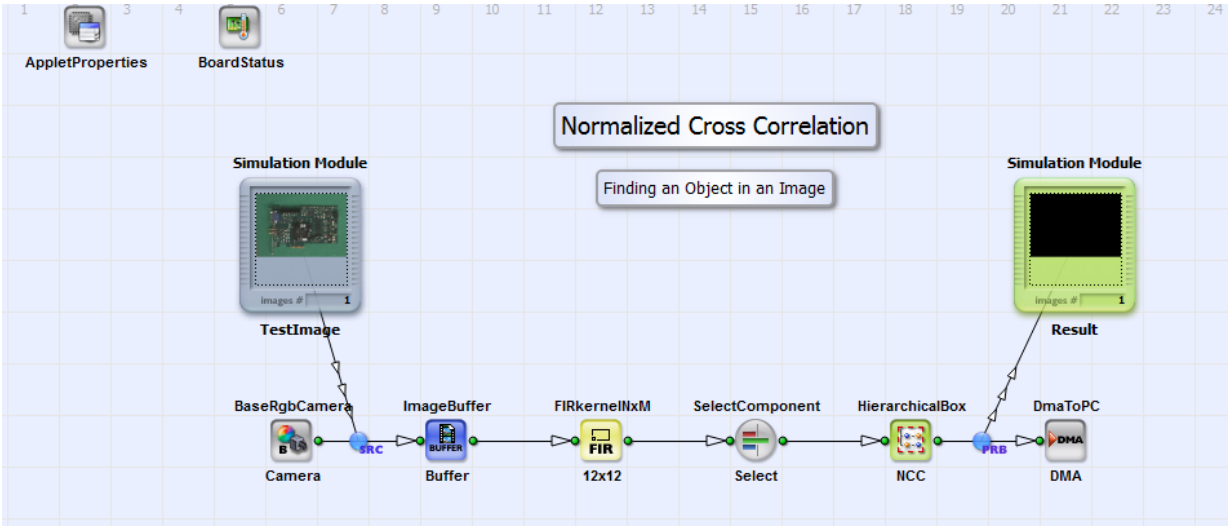



Figure 12.155. Basic design structure of "NormalizedCrossCorrelation.va"

In an image $I(x,y)$ from an RGB camera in Camera Link base configuration an image kernel of 12x12 pixels is defined with **FIRkernelNxM** operator **12x12**. The kernel size depends on the object to be identified. The object is identical to the reference image $R(u,v)$ in eq. 12.55. One color component (here: red) is selected with operator **SelectComponent** for identifying an object with the NCC algorithm according to eq. 12.55 in the HierarchicalBox **NCC**. As result a binary image with image size of original input image with value "1" at the identified object position and "0" at any other position is sent via DMA to PC. The example is configured for the RGB testimage "PCB.tif" which you find in the VisualApplets installation directory under examples\Processing\ObjectFeatures \NormalizedCrossCorrelation. This image has dimensions of 1024x712 pixels. In Fig. 12.156 the test image is shown.

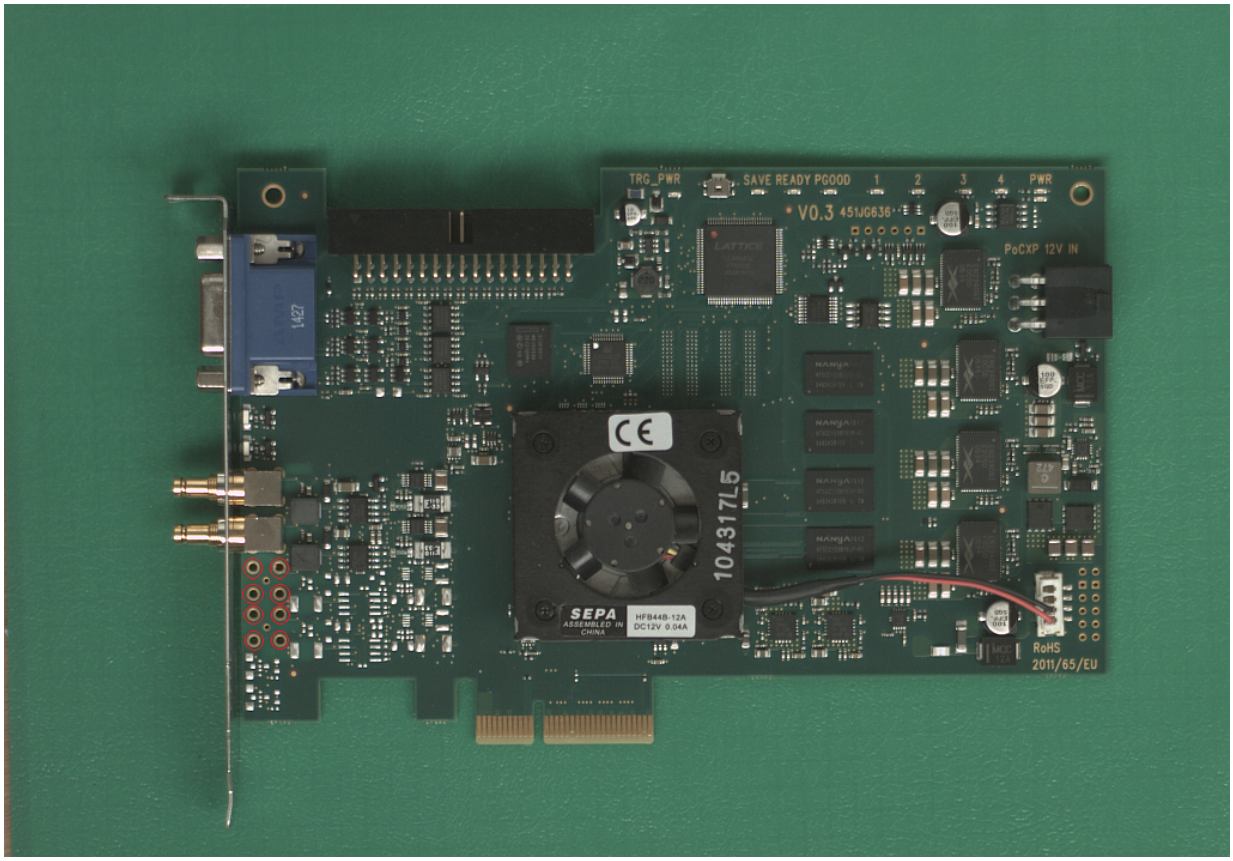



Figure 12.156. Test image "PCB.tif"

The six small holes on the lower left side of the PCB board are the objects to be identified in this example. The object dimensions are here 12x12 pixels. In the design in box **NCC** (see Fig. 12.155) the algorithm according to eq. 12.55 for locating these objects is implemented. Please see the comment boxes beside the links and HierarchicalbBoxes for further information on the corresponding equation part. In Fig. 12.157 you can see the content of HierarchicalBox **NCC**.

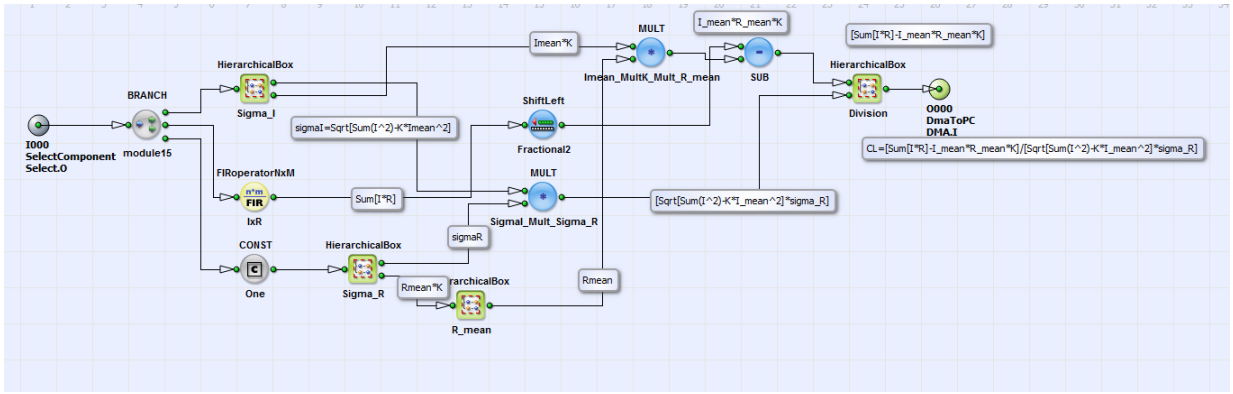



Figure 12.157. Content of HierarchicalBox **NCC**

You can adapt the design example to your image and objects of choice as explained in the following. If you want to use grayscale images instead of RGB images just replace operator **BaseRgbCamera** with operator **BaseGrayCamera** and delete operator **SelectComponent**.

1. Adapt the size of operator **FIRkernelNXM_12x12** to the dimensions of the objects you want to identify in the image.
2. With "right-mouse-click" on HierarchicalBox **NCC** under "Properties" you can set the number "K" of pixels of the object to be found. With object size of 12x12, K is 144. Under "Properties" you

can also adapt the binarization threshold. This threshold is used to visualize the objects position with "1" in the result image.

3. Adapt the number of kernel columns and rows to the object dimension at the output of **CONST** operator **One** in HierarchicalBox **NCC**.

4. Load pixel value file of the object to be identified (which is $R(u,v)$) to **FIRoperatorNxM IxR** in HierarchicalBox **NCC**. You can use program "NCC.m" to create this file ("R.txt").

5. Load the same file to the **FIRoperatornxM Rmean_Mult_K** in HierarchicalBox **Sigma_R** in HierarchicalBox **NCC**.

6. Load the file "Rpower2.txt" (created with Matlab program "NCC.m") to the **FIRoperatorNxM Sum_R_R** in HierarchicalBox **Sigma_R**. The content of this file are the pixel values of the object (divided by 2) to the power of two $(R/2)^2$.

In the HierarchicalBox **Division** (see Fig. 12.157) the final result $C_L(x,y)$ of eq. 12.55 is calculated with the division operation **DIV**. You can see the content of box **Division** in Fig. 12.158.

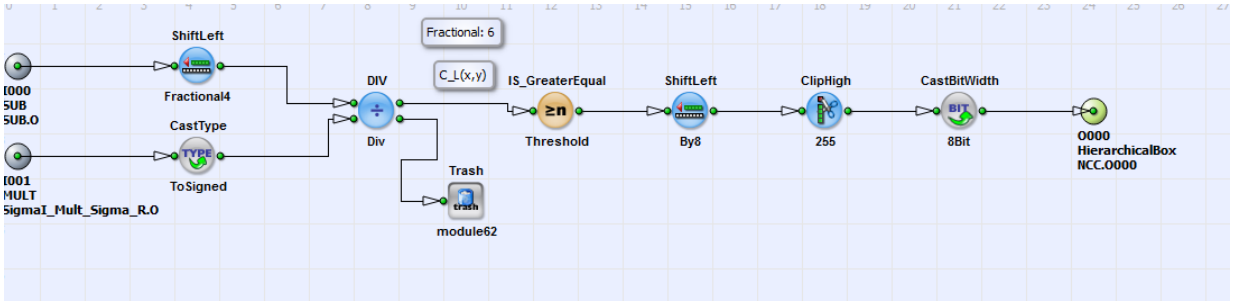



Figure 12.158. Content of HierarchicalBox **Division**

The division is performed with six fractional bits i.e. the result of $C_L(x,y)$ is between -64 and 64 instead of - 1 and 1. To visualize the position of the identified object the result image is binarized with **IS_GreaterEqual** operator **Threshold**. The threshold value is set to 50 in this example. You can vary this value directly in the operator or as explained above with "right-mouse-click" on box **NCC**. The result image which is sent to PC is shown in Fig. 12.159. The pixels at position of the small holes on the lower left side of the PCB board (see Fig. 12.156) obtain value "1" and all other pixels "0". The result image is sent via DMA to PC. Instead of this it can also be input for further image processing steps.

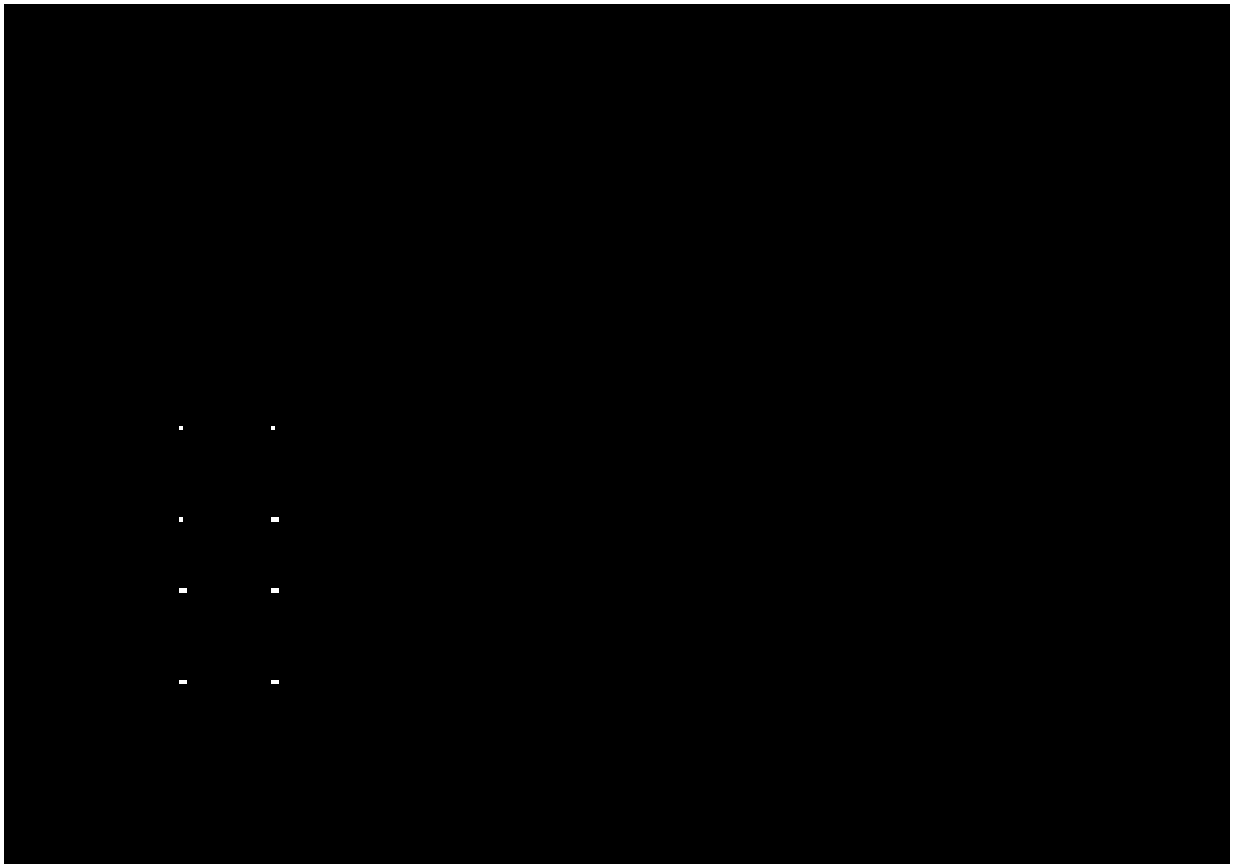
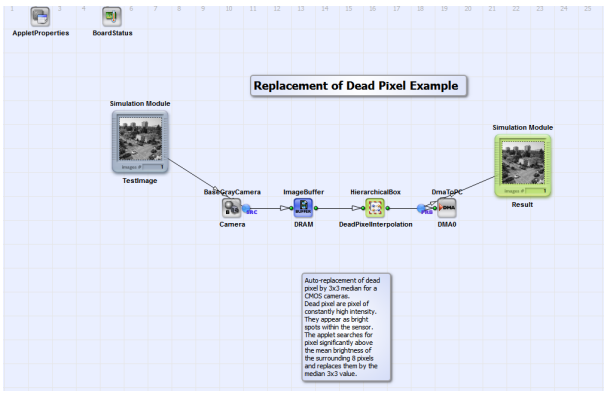


Figure 12.159. result image with "1" at object positions (zoomed view)

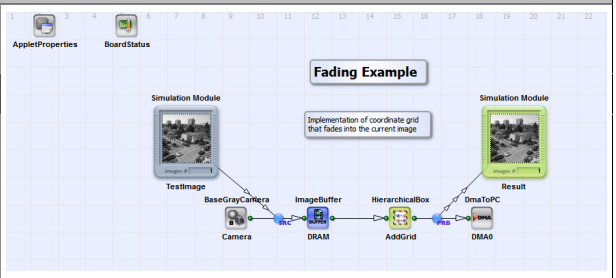
## 12.14. Shading Correction

Find in the following subsections examples on shading correction. We provide examples for shading correction in one and two dimensions. Also one VisualApplets design demonstrating the fading of one coordinate grid into the current image is implemented. A dead pixel replacement example is presented using a median filter.

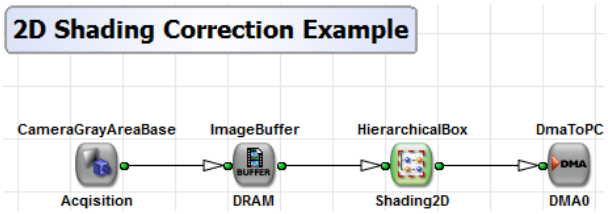
### 12.14.1. Dead Pixel Replacement

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Shading \DeadPixelReplace\DeadPixelReplace.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>The examples shows an automatic dead pixel detection and replacement. | |

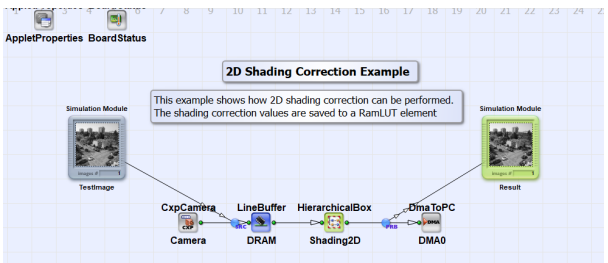
## 12.14.2. Grid Overlay Fading

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Shading\Fading \Fading.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A grid is overlayed to the input images. The grid pixel value is determined from the input pixel value. | |

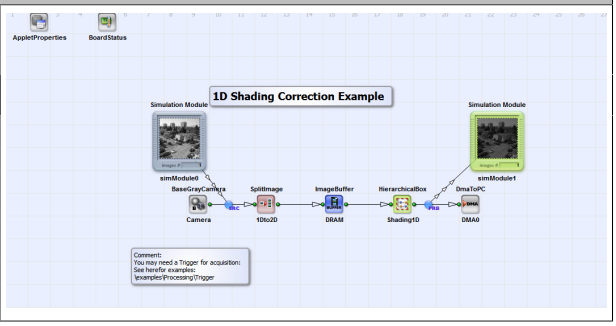
## 12.14.3. 2D Shading Correction / Flat Field Correction

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Shading\Shading2D \Shading2D.va |  |
| **Default Platform:** mE4VD4-CL | |
| **Short Description**<br><br>The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction. | |

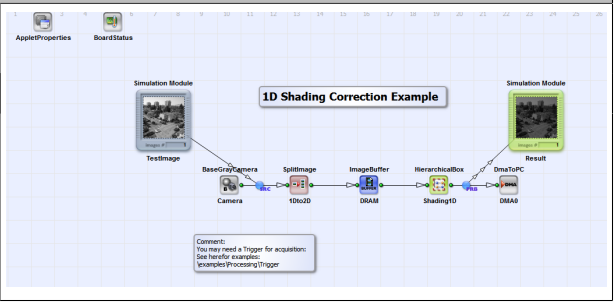
## 12.14.4. 2D Shading Correction / Flat Field Correction Using Operator RamLUT

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Shading\Shading2D \imaFlex_Shading2D_RAMLUT.va |  |
| **Default Platform:** iF-CXP12-Q | |
| **Short Description**<br><br>The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction. The design is implemented for the imaFlex CXP12 platform but can easily be tranferred and adapted to the microEnable5 frame grabbers. | |

## 12.14.5. 1D Shading Correction Using Block RAM

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Shading \Shading1D_BRAM\Shading1D_BRAM.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>The example shows an 1D shading correction. The correction values are stored in block RAM memory. |  |

### 12.14.6. 1D Shading Correction Using Frame Grabber RAM

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Shading \Shading1D_DRAM\Shading1D_DRAM.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM. |  |

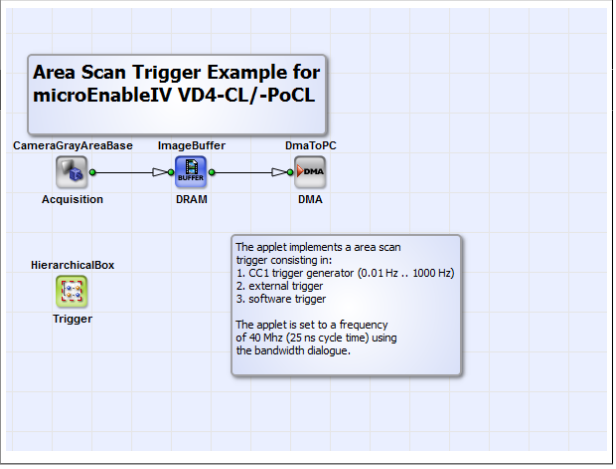
## 12.15. Trigger

Examples for area and line scan application. Usage of the signal processing operators.

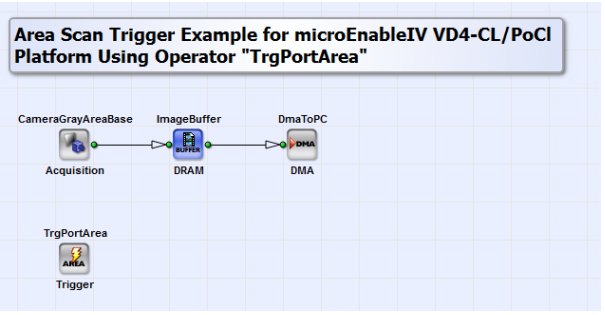
### 12.15.1. Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform

In the following subsections three example implementations for an area scan trigger for microEnable IV VD4-CL/-PoCL platform are presented.

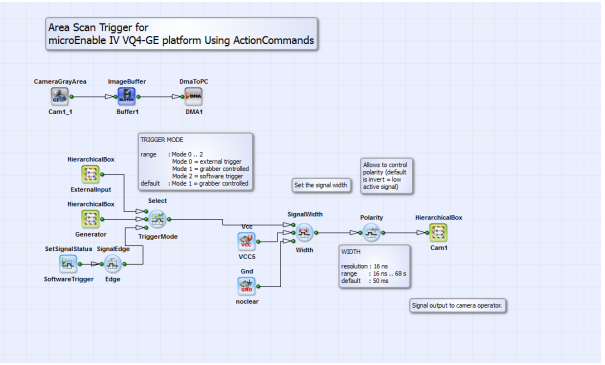
#### 12.15.1.1. Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator CameraControl

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE4VD4-CL \Area\AreaScanTrigger_mE4VD4CL.va<br>**Default Platform:** mE4VD4-CL/-PoCL<br>**Short Description**<br>An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The operator **CameraControl** provides the interface to the connected camera. |  |

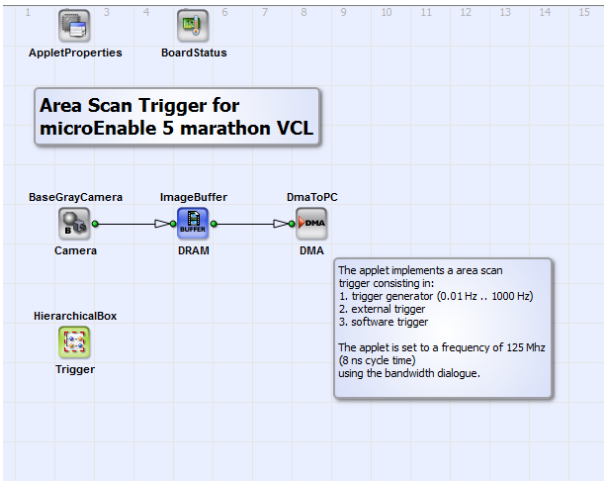
### 12.15.1.2. Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Operator TrgPortArea

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE4VD4-CL \Area\TriggerPortArea_mE4VD4CL.va<br>**Default Platform:** mE4VD4-CL/-PoCL<br>**Short Description**<br>An area scan trigger using the operator **TrgPortArea** is presented. Please read for information on this operator Section 29.49, 'TrgPortArea'. |  |

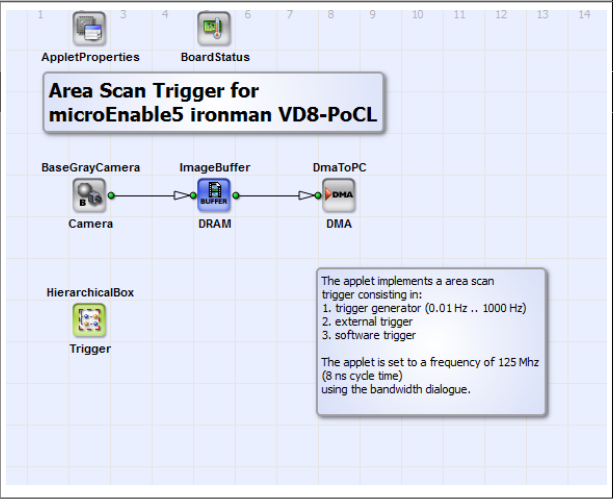
### 12.15.2. Area Scan Trigger for microEnable IV VQ4-GE/-GPoE

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE4VQ4-GE \Area\AreaScanTrigger_mE4VQ4GE.va<br>**Default Platform:** mE4VQ4-GE/-PoGE<br>**Short Description**<br>An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe\ActionCommands in your Framegrabber SDK installation folder. |  |

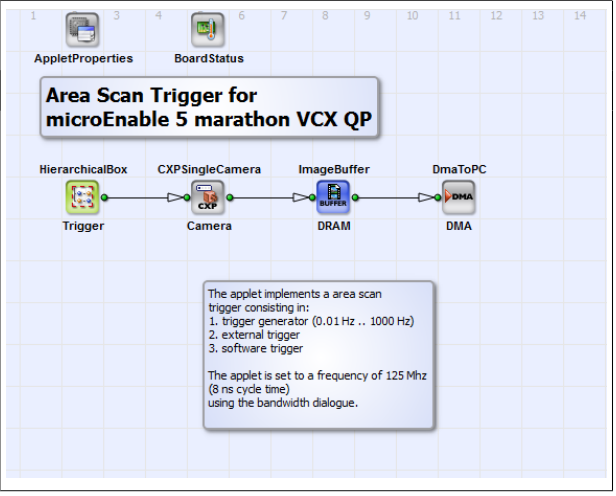
### 12.15.3. Area Scan Trigger for microEnable 5 marathon/LightBridge VCL

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5-MA-VCL \Area\AreaScanTrigger_mE5MAVCL.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. |  |

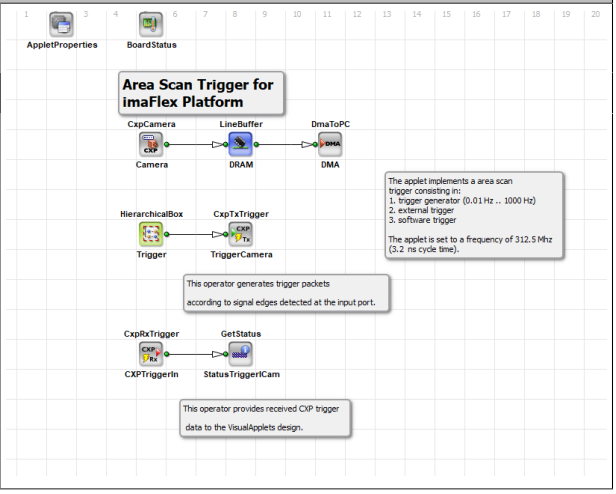
### 12.15.4. Area Scan Trigger for microEnable 5 VD8-CL/-PoCL

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5VD8-CL \Area\AreaScanTrigger_mE5VD8PoCL.va<br>**Default Platform:** mE5VD8-CL/-PoCL<br>**Short Description**<br>An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. |  |

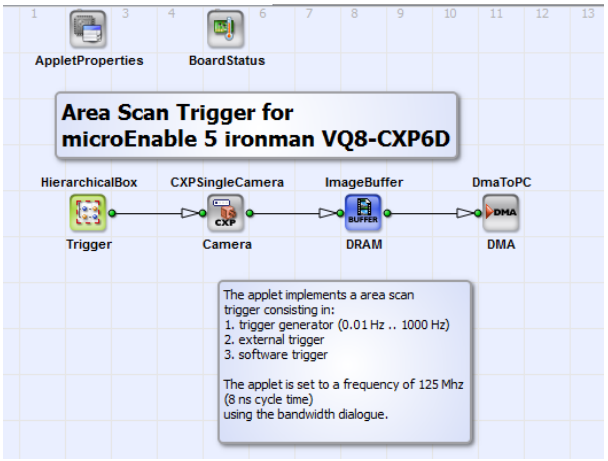
## 12.15.5. Area Scan Trigger for microEnable 5 marathon VCX QP

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5-MA-VCX-QP\Area\AreaScanTrigger_mE5VCXQP.va<br>**Default Platform:** mE5-MA-VCX-QP<br>**Short Description**<br>An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. |  |

## 12.15.6. Area Scan Trigger for imaFlex CXP-12 Quad

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\iF-CXP12-Q \Area\AreaScanTrigger_iFCXP12Q.va<br>**Default Platform:** iF-CXP12-Q<br>**Short Description**<br>An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. |  |

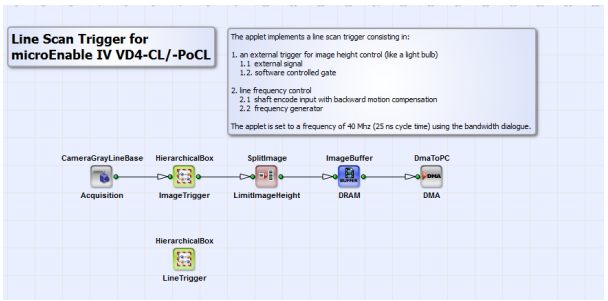
## 12.15.7. Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5VQ8-CXP\Area\AreaScanTrigger_mE5VQ8CXP6B.va<br>\examples\Processing\Trigger\mE5VQ8-CXP\Area\AreaScanTrigger_mE5VQ8CXP6D.va |  |
| **Default Platform:** mE5VQ8-CXP6D/mE5VQ8-CXP6B | |
| **Short Description**<br><br>An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. | |

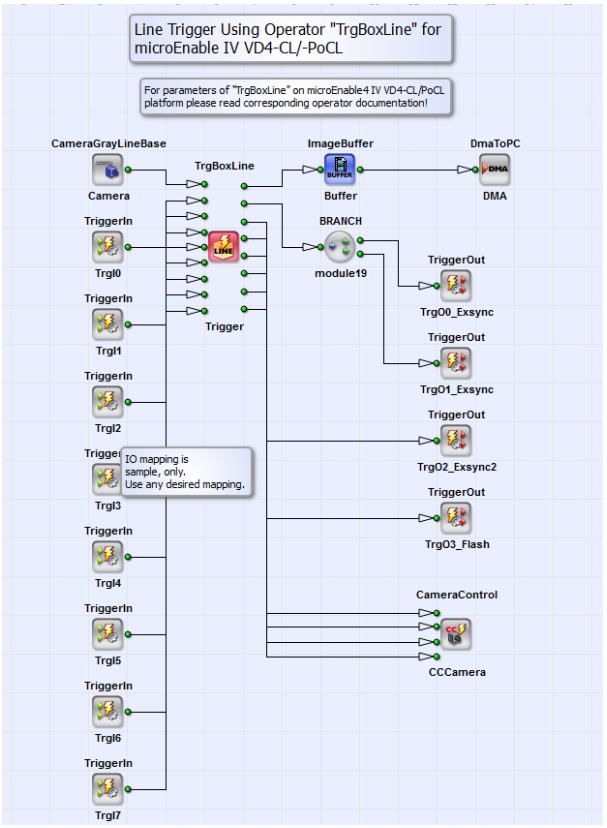
## 12.15.8. Line Scan Trigger for microEnable IV VD4-CL/-PoCL

In the following three example implementations for line scan trigger for microEnable IV VD4-CL/-PoCL platform are presented.

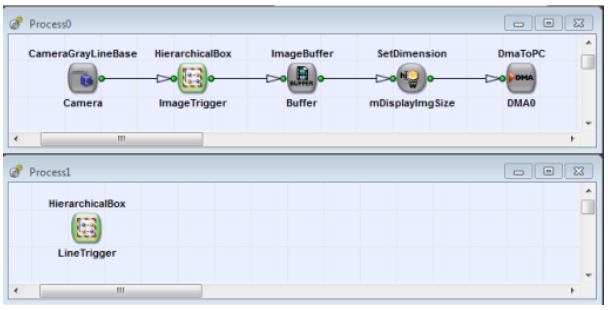
### 12.15.8.1. Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator CameraControl

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE4VD4-CL\Line\LineScanTrigger_mE4VD4CL.va |  |
| **Default Platform:** mE4VD4-CL/-PoCL | |
| **Short Description**<br><br>A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. | |

### 12.15.8.2. Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Operator TrgBoxLine

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE4VD4-CL\Line\TrgBoxLine_mE4VD4CL.va<br>**Default Platform:** mE4VD4-CL/-PoCL<br>**Short Description**<br><br>VisualApplets prototype operator *TrgBoxLine* can be used for microEnable 5 and microEnable IV platforms for line scan triggering. The operator includes multiple triggering features. The example demonstrates the usage of the operator in a VisualApplets design file for microEnable IV VD4-CL/PoCL. For parameters of "TrgBoxLine" on microEnable IV VD4-CL/PoCL platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine'. Note that in comparison to operator *TrgPortLine* the operator does not directly control the IOs. Therefore, it is required to connect to the IOs respectively.<br><br>The operator is in prototype library and might be replaced in future releases. |  |

### 12.15.8.3. Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE4VD4-CL\Line\TrgPortLineRebuild_mE4VD4CL.va<br>**Default Platform:** mE4VD4-CL<br>**Short Description**<br><br>VisualApplets operator *TrgPortLine* is rebuild with other signal processing operators from the VisualApplets operator libraries. The rebuild allows you custom trigger functionalities and shows the usage of many signal processing operators in VisualApplets. |  |

Operator *TrgPortLine* is a complex VisualApplets operator which offers a wide range of line scan trigger functionalities. In some applications it is required to have a modified functionality of this operator for custom signal processing.

In this example, we present a rebuild of the *TrgPortLine* operator with other VisualApplets signal processing operators. This enables you to adapt and modify the functionality to your custom requirements. The implementation is advanced but easy to understand. To simply the complex parameterization of the implementation, the following translation table helps you to calculate the parameters of the new design based on the parameternames of the *TrgPortLine* parameters.

| Name | YOffset |
|---|---|
| Default | 0 |

```
Device1_Process0_ImageTrigger_ImageHeight_Width = YHeight - YOffset
          Device1_Process0_ImageTrigger_ImageTriggerDelay_Delay_Delay = YHeight + YOffset
```

| Name | YHeight |
|---|---|
| Default | 1024 |

```
Device1_Process0_ImageTrigger_ImageHeight_Width = YHeight - YOffset
          Device1_Process0_ImageTrigger_ImageTriggerDelay_Delay_Delay = YHeight + YOffset
```

| Name | LineTriggerMode |
|---|---|
| Default | GrabberControlled |

```
Device1_Process1_LineTrigger_LineTriggerMode_Select = 3 if GrabberController 0,1 or 2 if Extern_Trigger
          For Extern_Trigger, the correct value is defined by the shaft encoder settings.
          The other modes available by TrgPortLine are not supported.
```

| Name | ExsyncEnable |
|---|---|
| Default | OFF |

```
Device1_Process1_LineTrigger_Exsync_ExsyncEnable_Select = 1 if ON  0 if OFF
```

| Name | LineTrgInSourceA |
|---|---|
| Default | 0 |

```
Device1_Process1_LineTrigger_TraceA_Input_Select = LineTrgInSourceA
```

| Name | LineTrgInSourceB |
|---|---|
| Default | 1 |

```
Device1_Process1_LineTrigger_TraceB_Input_Select = LineTrgInSourceB
```

| Name | EncoderABMode |
|---|---|
| Default | Signal_A_Only |

```
Device1_Process1_LineTrigger_LineTriggerMode_Select = 3 if LineTriggermode == GrabberControlled
            0 if EncoderABMode == Signal_A_Only 1 else Device1_Process1_LineTrigger_ShaftEncoder_Mode =
            Mode1X if EncoderABMode == Signal_AB_Filter
            Mode2X if EncoderABMode == Signal_ABx2_Filter
            Mode4X if EncoderABMode == Signal_ABx4_Filter
```

| Name | EncoderABLead |
|---|---|
| Default | Signal_AB |

```
Device1_Process1_LineTrigger_ShaftEncoder_LeadingTrace =
           A if EncoderABLead == Signal_AB
           B else
```

| Name | LineTrgInPolarity |
|---|---|
| Default | LowActive |

```
Device1_Process1_LineTrigger_TraceA_Polarity_Invert =
            Invert if LineTrgInPolarity = LowActive
            NotInvert else
```

| Name | LineTrgInPolarity |
|---|---|
| Default | LowActive |

```
Device1_Process1_LineTrigger_TraceA_Polarity_Invert =
            Invert if LineTrgInPolarity = LowActive
            NotInvert else
          Device1_Process1_LineTrigger_TraceB_Polarity_Invert =
            Invert if LineTrgInPolarity = LowActive
            NotInvert else
```

| Name | LineTrgDownscaler |
|---|---|
| Default | 1 |

```
Device1_Process1_LineTrigger_Downscale_Donwscale = LineTrgDownscaler
```

| Name | LineTrgPhase |
|---|---|
| Default | 1 |

```
Device1_Process1_LineTrigger_Downscale_SelectecPulse = LineTrgPhase
```

| Name | ExsyncPeriod |
|---|---|
| Default | 100µs |

```
Device1_Process1_LineTrigger_Generate_Period = ExsyncPeriod / TClk
```

| Name | Exsync2Delay |
|---|---|
| Default | 0µs |

```
Device1_Process1_LineTrigger_Exsync_Exsync2_Delay = Exsync2Delay / TClk
```



**VisualApplets Version 2.1 or Higher Only**

The delay has to be set to 0 for VisualApplets version 2.0.

| Name | ExsyncPolarity |
|---|---|
| Default | LowActive |

```
Device1_Process1_LineTrigger_Exsync_OutputPolarity_Polarity =
            Invert if ExsyncPolarity = LiwActive
           NotInvert else
```

| Name | ImgTriggerMode |
|---|---|
| Default | FreeRun |

```
Device1_Process0_ImageTrigger_SelectGatedImageTrigger_Select =
           1 if ImgTriggerMode == FreeRun
            0 else
           Device1_Process0_ImageTrigger_SelectFreeRun_Select =
             1 if ImgTriggerMode == FreeRun
            0 else
```

| Name | ImgTriggerInSource |
|---|---|
| Default | InSignal0 |

```
Device1_Process0_ImageTrigger_EnableSoftwareTrigger =
            1 if ImgTriggerInSource == SoftwareTrigger
           0 else
          Device1_Process0_ImageTrigger_ImageTriggerInput_Input_Select =
            0 if ImgTriggerInSource == InSignal0
            1 if ImgTriggerInSource == InSignal1
            2 if ImgTriggerInSource == InSignal2
            3 if ImgTriggerInSource == InSignal3
            4 if ImgTriggerInSource == InSignal4
            5 if ImgTriggerInSource == InSignal5
            6 if ImgTriggerInSource == InSignal6
            7 if ImgTriggerInSource == InSignal7
```

| Name | ImgTrgInPolarity |
|---|---|
| Default | LowActive |

```
Device1_Process0_ImageTrigger_ImageTriggerInput_Polarity_Polarity =
            Invert if ImgTrgInPolarity == LowActive
            NotInvert else
```

| Name | ImgTrgDelay |
|---|---|
| Default | 0 lines |

```
Device1_Process0_ImageTrigger_ImageTriggerDelay_Delay_Delay = ImgTrgDelay
```



**VisualApplets Version 2.1 or Higher Only**

The delay has to be set to 0 for VisualApplets version 2.0.

| Name | FlashEnable |
|---|---|
| Default | OFF |

```
Device1_Process0_ImageTrigger_FlashOutput_Enable_Select =
          1 if FlashEnable == ON
          0 else
```

| Name | FlashPolarity |
|---|---|
| Default | LowActive |

```
Device1_Process0_ImageTrigger_FlashOutput_Polarity_Polarity =
          Invert if FlashPolarity == LowActive
          NotInvert else
```

| Name | FlashDelay |
|---|---|
| Default | 0 lines |

```
Device1_Process0_ImageTrigger_FlashDelay_Delay_Delay = FlashDelay
```



**VisualApplets Version 2.1 or Higher Only**

The delay has to be set to 0 for VisualApplets version 2.0.

| Name | SoftwareTrgPulse |
|---|---|

```
Device1_Process0_ImageTrigger_SoftwareTrigger_Mode =
           Pulse if SoftwareTrgPulse == 1 && ImgTrgMode == ExternSw_Trigger
            not write access else
```

| Name | SoftwareTrgInput |
|---|---|

```
Device1_Process0_ImageTrigger_SoftwareTrigger_Mode =
           High if SoftwareTrgInput == 1 && ImgTrgMode == ExternSw_Gate
             Low else
```

| Name | ImgTrgIsBusy (read only parameter) |
|---|---|

```
ImgTrgIsBusy = Device1_Process0_ImageTrigger_ImgTrgIsBusy_Status
```

| Name | CC1output |
|---|---|
| Default | Exsync |

```
Device1_Process1_LineTrigger_Exsync_CC1_Select_Select =
           0 if Exsync
           1 if ExsyncInvert
           2 if Exsync2
           3 if Exsync2Invert
           4 if Flash
           5 if FlashInvert
           6 if Gnd
           7 if Vcc
```

| Name | CC2output |
|---|---|
| Default | Exsync |

```
Device1_Process1_LineTrigger_Exsync_CC2_Select_Select =
           0 if Exsync
           1 if ExsyncInvert
           2 if Exsync2
           3 if Exsync2Invert
           4 if Flash
           5 if FlashInvert
           6 if Gnd
           7 if Vcc
```

| Name | CC3output |
|---|---|
| Default | Exsync |

```
Device1_Process1_LineTrigger_Exsync_CC3_Select_Select =
           0 if Exsync
           1 if ExsyncInvert
           2 if Exsync2
           3 if Exsync2Invert
           4 if Flash
           5 if FlashInvert
           6 if Gnd
           7 if Vcc
```

| Name | CC4output |
|---|---|
| Default | Exsync |

```
Device1_Process1_LineTrigger_Exsync_CC4_Select_Select =
           0 if Exsync
           1 if ExsyncInvert
           2 if Exsync2
           3 if Exsync2Invert
           4 if Flash
           5 if FlashInvert
           6 if Gnd
           7 if Vcc
```

| Name | ImgTrgDebounceMaxTime |
|---|---|
| Not Required in this implementation. | |

| Name | ImgTrgDebouncingTime |
|---|---|
| Default | 65.520µs |

```
Device1_Process0_ImageTrigger_ImageTriggerInput_Debounce_Debounce = ImgTrgDebouncingTime / TClk
```

| Name | LineTrgDebouncingTime |
|---|---|
| Default | 0.112µs |

```
Device1_Process1_LineTrigger_TraceA_Debounce_Debounce = LineTrgDebouncingTime / TClk
          Device1_Process1_LineTrigger_TraceB_Debounce_Debounce = LineTrgDebouncingTime / TClk
```

The variable TClk depends on the frame grabber used. For the microEnable IV VD4-CL, the period is 16ns. Check Appendix A, '*Device Resources*' for a complete list.

### 12.15.8.3.1. Implementation

The implementation is divided into the two parts ImageTrigger and LineTrigger. The LineTrigger implementation is located in Process1. As this process has no DMA channel, it will be immediately started after loading the applet to the frame grabber even when the acquisition is not started, yet. This has the advantage, that the camera can already be triggered before the acquisition is started. See Section 4.4.2, 'Processes without DMAs / Trigger Processes' for more information on processes without DMA channels.

The next two sections will outline the implementations of the image trigger and line trigger.

### 12.15.8.3.2. Image Trigger Implementation

The image trigger part is responsible to assemble the image i.e. to form 2D images from the 1D input line stream. In accordance to operator *TrgPortLine* the image height can be defined by either a fixed height and usage of all input lines (free run), a fixed height but external controller image start (external or software trigger) or the height can be defined by the pulse length of an external or software trigger signal (gated mode).

The main part of the image trigger implementation is the *SignalGate* operator. The image height is defined by the time the gate is open. Module *ImageHeight* (operator *SignalWidth*) defines the height in the free run or edge controlled trigger modes. In these cases, the operator value is increases with every new input line. This is a good example where the *Tick* input is used for controlling the counting speed. In the free run mode, a new image is started immediately after the previous one is finished i.e. the *SignalWidth* operator is finished with the output of its period. In the external or software trigger mode, a new image is started with a rising edge at the input of the *SignalWidth*. The RS-FF is required to ensure a sufficiently long enough pulse.

With *SelectGatedImageTrigger* the implementation can be switched to the gated mode, where the pulse length controls the image height.

The hierarchical box *NOP_DelayChain* includes some *NOP* operators. These operators are required to delay the input signal a few clock cycles. This is required as the calculation weather the gate has to be opened or closes requires some clock cycles. VisualApplets cannot perform an automated pipeline adjustment in this case. Therefore the delay is required.

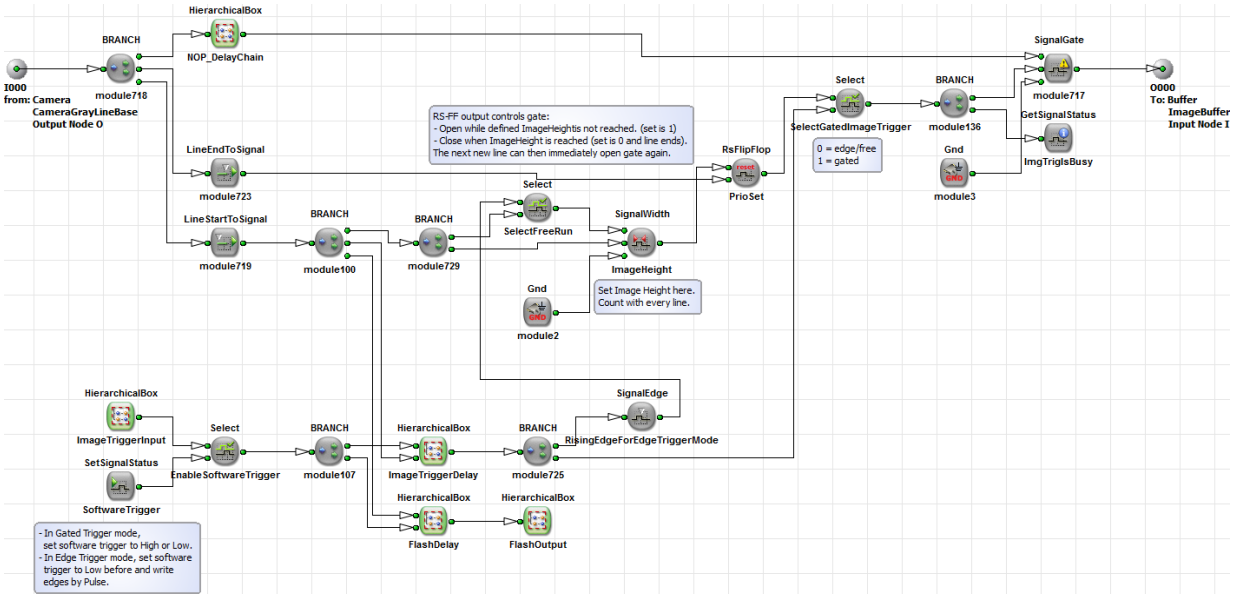



Figure 12.160. Hierarchical Box ImageTrigger of the TrgPortLine Rebuild Example

Other parst of the implementation comprise the input output output signal generation. The implementation of these parts is simple and straight forward. The flash output signal is transfered to Process1 using a *TxSignalLine* operator in h-box *FlashOutput*. This is required as the flash signal has to be output to the CC signals as well.

### 12.15.8.3.3. Line Trigger Implementation

The LineTrigger hierarchical box mainly consists of the input and output signal generation as well as the shaft encoder analysis and frequency generator. For the shaft encoder signal analysis operators *ShaftEncoder*, *ShaftEncoderCompensate* are used. The *Select* switch *LineTriggerMode* allows for an individual selection if the compensation or two trance analysis is enabled or disabled. Moreover, an internal frequency generator represented by a *Generate* operator is implemented to trigger the camera.

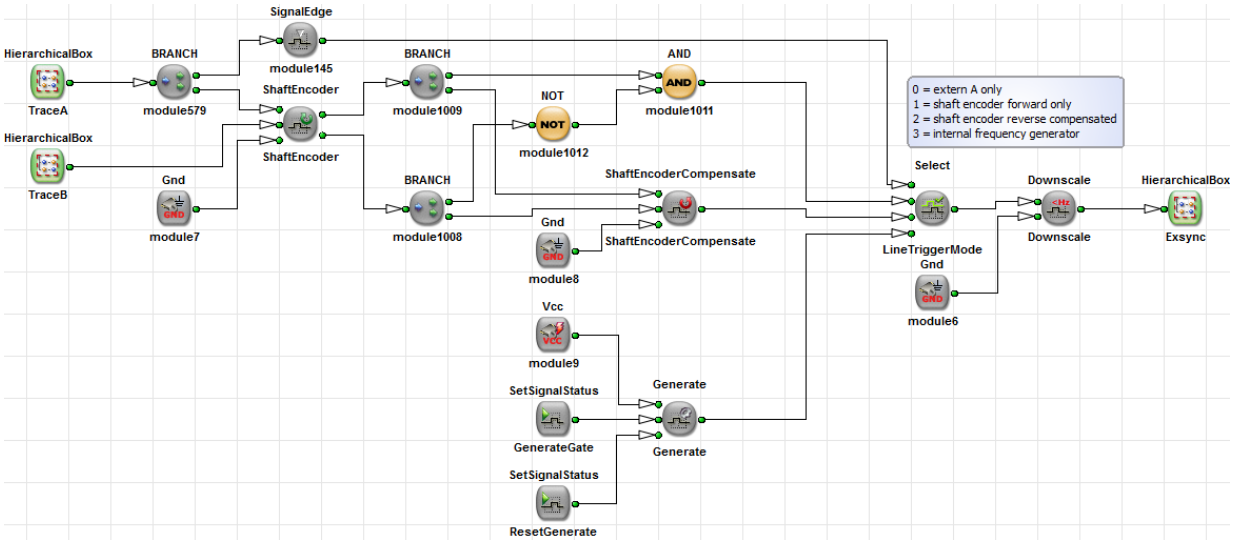
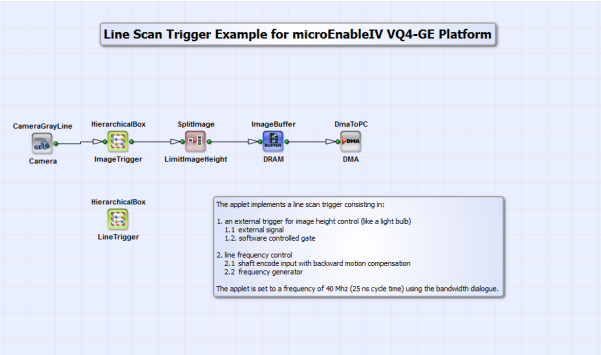



Figure 12.161. Hierarchical Box LineTrigger of the TrgPortLine Rebuild Example

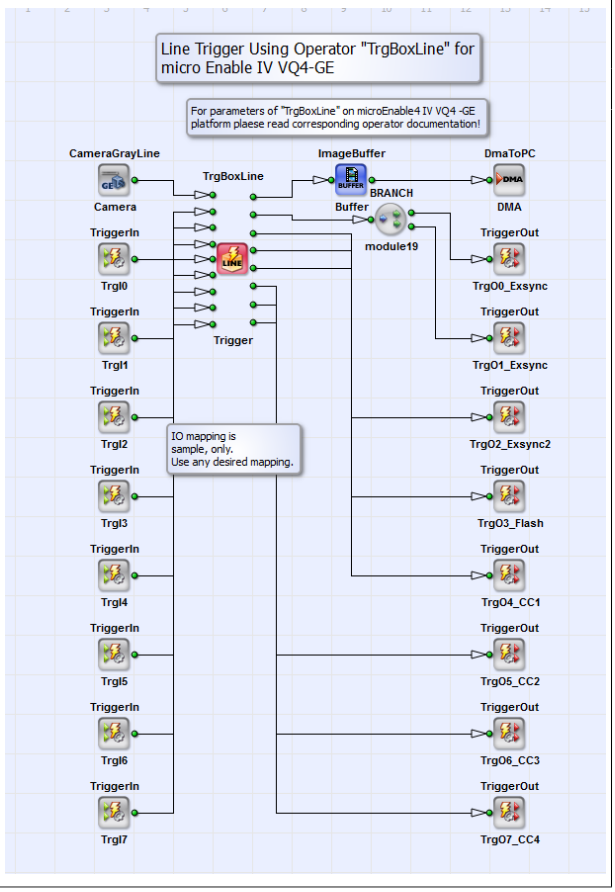
## 12.15.9. Line Scan Trigger for microEnable IV VQ4-GE/-GPoE

In the following two example implementations for a line scan trigger on a microEnable IV VQ4-GE/-GPoE platform are presented.

### 12.15.9.1. Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator ActionCommand

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Trigger\mE4VQ4GE\Line\LineScanTrigger_mE4VQ4GE.va |  |
| **Default Platform:** mE4VQ4-GE/-GPoE | |
| **Short Description**<br><br>A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe\ActionCommands in your Framegrabber SDK installation folder. | |

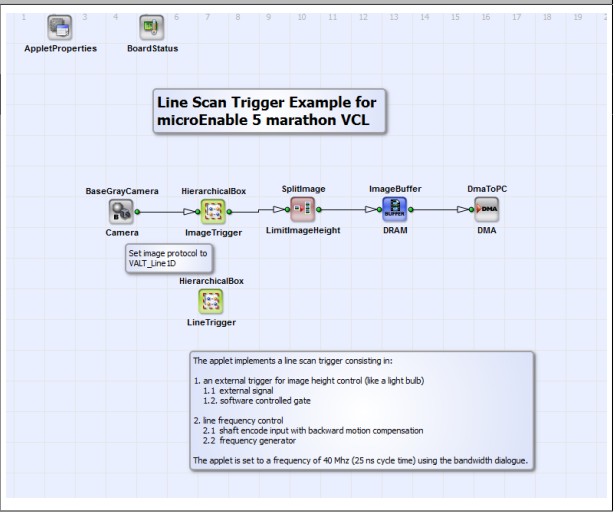
### 12.15.9.2. Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Operator TrgBoxLine

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE4VQ4-GE\Line\TrgBoxLine_mE4VQ4GE.va<br>**Default Platform:** mE4VQ4-GE/-GPoE<br>**Short Description**<br>VisualApplets prototype operator *TrgBoxLine* can be used for microEnable 5 and microEnable IV platforms for line scan triggering. The operator includes multiple triggering features. The example demonstrates the usage of the operator in a VisualApplets design file for microEnable IV VQ4-GE/GPoE. For parameters of "TrgBoxLine" on microEnable IV VQ4-GE/GPoE platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine'. Note that in comparison to operator *TrgPortLine* the operator does not directly control the IOs. Therefore, it is required to connect to the IOs respectively.<br>The operator is in prototype library and might be replaced in future releases. |  |

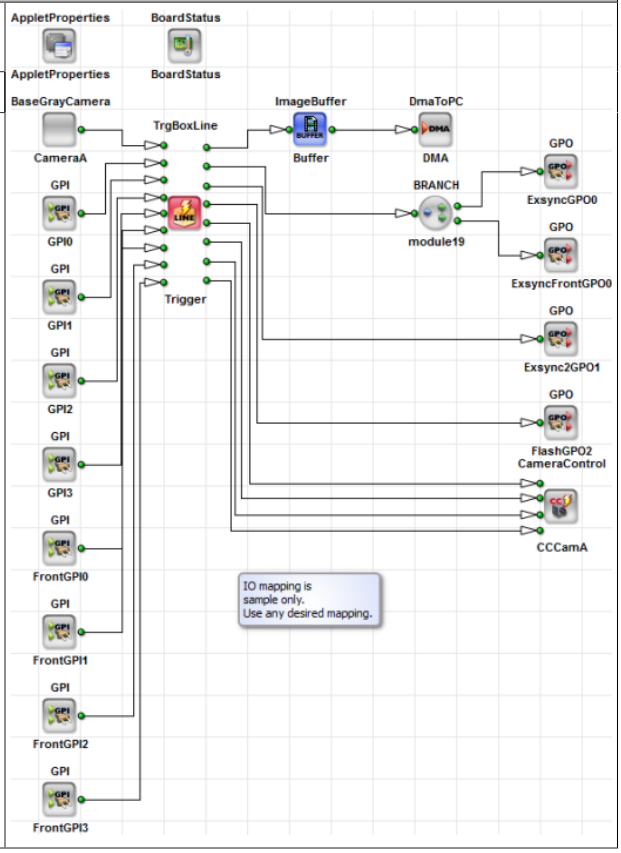
## 12.15.10. Line Scan Trigger for microEnable 5 marathon/LightBridge VCL

In the following two example implementations for a line scan trigger on a microEnable 5 marathon/ LightBridge VCL platform are presented.

### 12.15.10.1. Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator CameraControl

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5-MA-VCL\Line\LineScanTrigger_mE5MAVCL.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. |  |

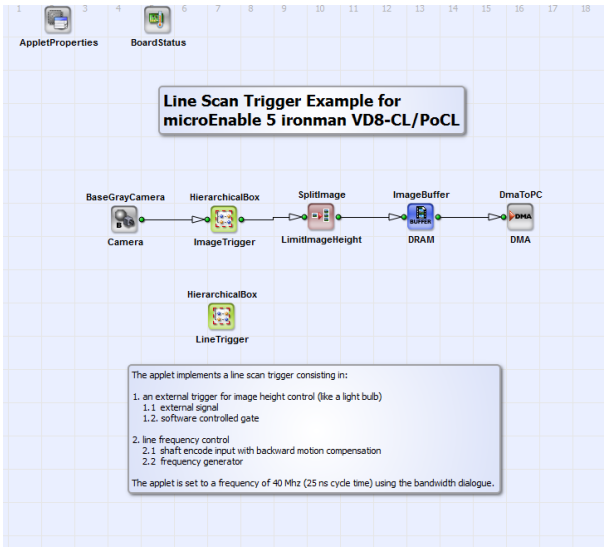
### 12.15.10.2. Line Scan Trigger for microEnable 5 marathon/LightBridge VCL with TrgBoxLine Operator Usage

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5-MA-VCL\Line\TrgBoxLine_mE5MAVCL.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>VisualApplets prototype operator *TrgBoxLine* can be used for microEnable 5 and microEnable IV platforms for line scan triggering. The operator includes multiple triggering features. The example demonstrates the usage of the operator in a VisualApplets design file. Note that in comparison to operator *TrgPortLine* the operator does not directly control the IOs. Therefore, it is required to connect to the IOs respectively.<br>The operator is in prototype library and might be replaced in future releases. |  |

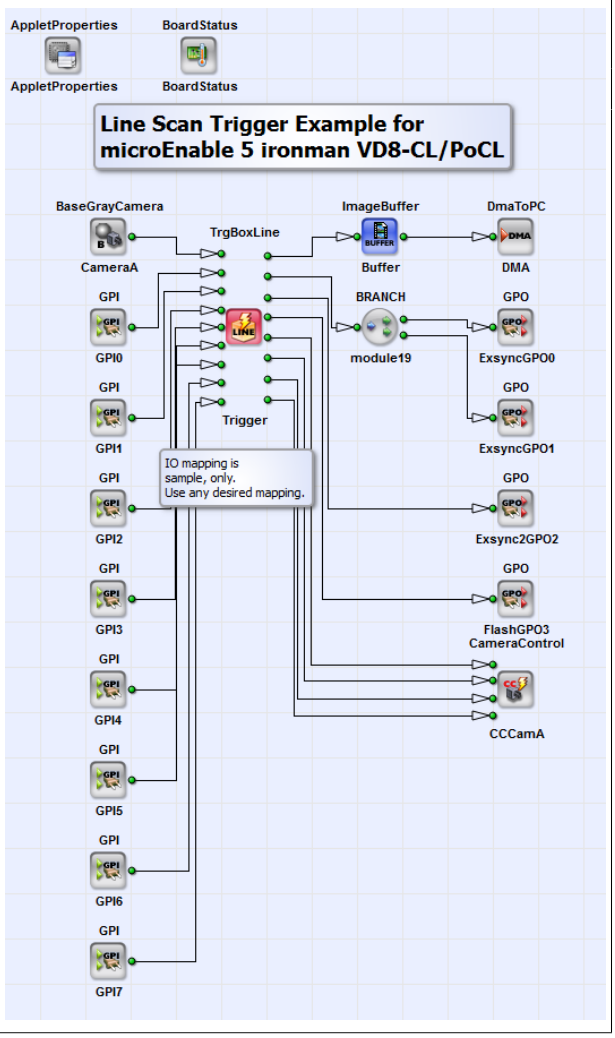
### 12.15.11. Line Scan Trigger for microEnable 5 VD8-CL/-PoCL

In the following two example implementations for a line scan trigger on a microEnable 5 VD8-CL/-PoCL platform are presented.

### 12.15.11.1. Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator CameraControl

| **Brief Description** | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5VD8-CL \Line\LineScanTrigger_mE5VD8CL.va |  |
| **Default Platform:** mE5VD8-CL/-PoCL | |
| **Short Description**<br><br>A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. | |

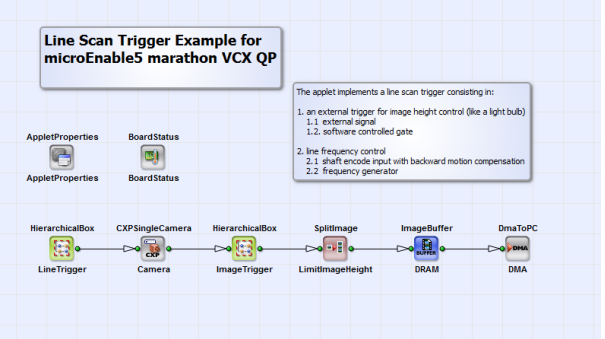
### 12.15.11.2. Line Scan Trigger for microEnable 5 VD8-CL/-PoCL with TrgBoxLine Operator Usage

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5VD8-CL\Line\TrgBoxLine_mE5VD8CL.va<br>**Default Platform:** mE5VD8-CL/-PoCL<br>**Short Description**<br>VisualApplets prototype operator *TrgBoxLine* can be used for microEnable 5 and microEnable IV platforms for line scan triggering. The operator includes multiple triggering features. The example demonstrates the usage of the operator in a VisualApplets design file for mE5VD8-CL/-PoCL platform. For parameters of "TrgBoxLine" on this platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine' Note that in comparison to operator *TrgPortLine* on mE4VD4-CL/PoCL the operator does not directly control the IOs. Therefore, it is required to connect to the IOs respectively.<br>The operator is in prototype library and might be replaced in future releases. |  |

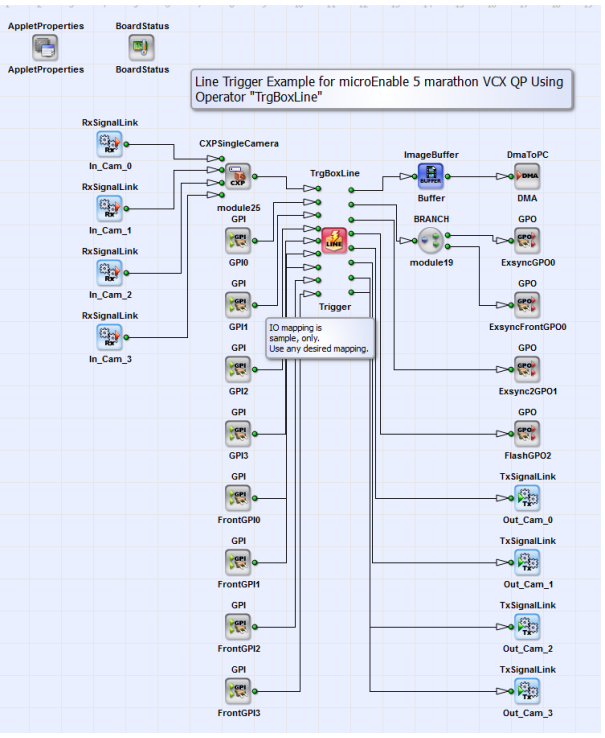
## 12.15.12. Line Scan Trigger for microEnable 5 marathon VCX QP

In the following two example implementations for a line scan trigger on a microEnable 5 marathon VCX QP platform are presented.

### 12.15.12.1. Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5-MA-VCX-QP\Line\LineScanTrigger_mE5MAVCXQP.va<br>**Default Platform:** mE5-MA-VCX-QP<br>**Short Description**<br>A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. |  |

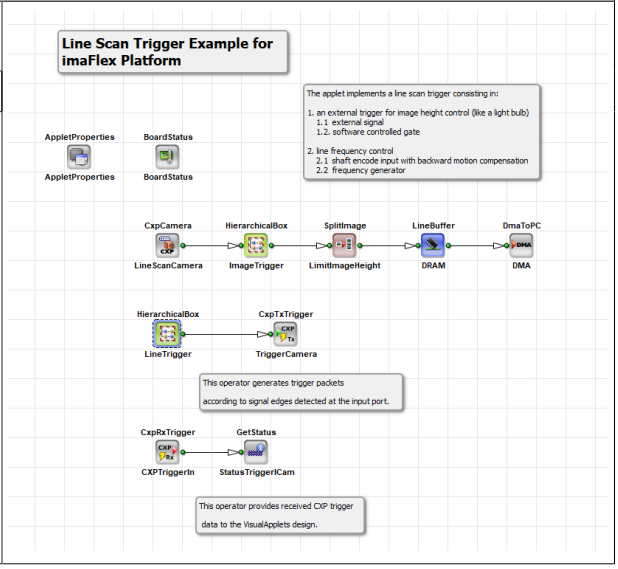
### 12.15.12.2. Line Scan Trigger for microEnable 5 marathon VCX QP with TrgBoxLine Operator Usage

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5-MA-VCX-QP\Line\TrgBoxLine_mE5MAVCXQP.va<br>**Default Platform:** mE5-MA-VCX-QP<br>**Short Description**<br>VisualApplets prototype operator *TrgBoxLine* can be used for for line scan triggering. The operator includes multiple triggering features. The example demonstrates the usage of the operator in a VisualApplets design file for mE5-MA-VCX-QP platform. For parameters of "TrgBoxLine" on this platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine' Note that in comparison to operator *TrgPortLine* on mE4VD4-CL/PoCL the operator does not directly control the IOs. Therefore, it is required to connect to the IOs respectively. |  |

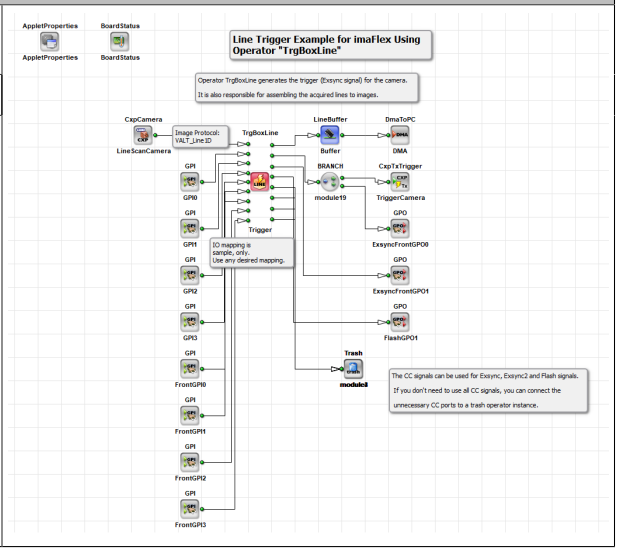
### 12.15.13. Line Scan Trigger for imaFlex CXP-12 Quad

In the following two example implementations for a line scan trigger on an imaFlex CXP-12 Quad platform are presented.

### 12.15.13.1. Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\iF-CXP12-Q \Line\LineScanTrigger_iFCXP12Q.va<br>**Default Platform:** iF-CXP12-Q<br>**Short Description**<br>A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. |  |

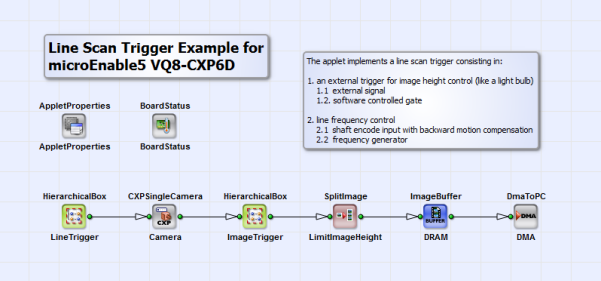
### 12.15.13.2. Line Scan Trigger for imaFlex CXP-12 Quad with TrgBoxLine Operator Usage

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\iF-CXP12-Q \Line\TrgBoxLine_iFCXP12Q.va<br>**Default Platform:** iF-CXP12-Q<br>**Short Description**<br>VisualApplets operator *TrgBoxLine* can be used for for line scan triggering. The operator includes multiple triggering features. The example demonstrates the usage of the operator in a VisualApplets design file for imaFlex CXP-12 Quad platform. For parameters of "TrgBoxLine" please read corresponding operator documentation under Section 30.5, 'TrgBoxLine' |  |

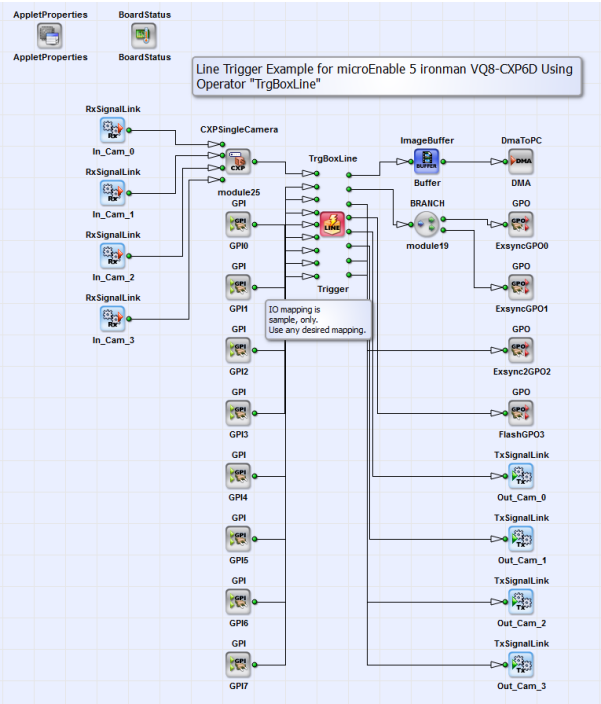
## 12.15.14. Line Scan Trigger for microEnable 5 VQ8-CXP6

In the following two example implementations for a line scan trigger on microEnable 5 VQ8-CXP6B and microEnable 5 VQ8-CXP6D platforms are presented.

### 12.15.14.1. Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5VQ8-CXP6D\Line\LineScanTrigger_mE5VQ8CXP6D.va \examples\Processing\Trigger\mE5VQ8-CXP6B\Line\LineScanTrigger_mE5VQ8CXP6D.va |  |
| **Default Platform:** mE5VQ8-CXP6D/mE5VQ8-CXP6B | |
| **Short Description**<br><br>A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. | |

### 12.15.14.2. Line Scan Trigger for microEnable 5 VQ8-CXP6 with TrgBoxLine Operator Usage

| Brief Description | |
|---|---|
| **File:** \examples\Processing\Trigger\mE5-VQ8-CXP6D\Line\TrgBoxLine_mE5VQ8CXP6D.va \examples\Processing\Trigger\mE5-VQ8-CXP6B\Line\TrgBoxLine_mE5VQ8CXP6B.va |  |
| **Default Platform:** mE5-VQ8-CXP6D/mE5-VQ8-CXP6B | |
| **Short Description**<br><br>VisualApplets prototype operator *TrgBoxLine* can be used for microEnable 5 and microEnable IV platforms for line scan triggering. The operator includes multiple triggering features. The example demonstrates the usage of the operator in a VisualApplets design file for mE5-VQ8-CXP6D/mE5-VQ8-CXP6B platform. For parameters of "TrgBoxLine" on this platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine' Note that in comparison to operator *TrgPortLine* on mE4VD4-CL/PoCL the operator does not directly control the IOs. Therefore, it is required to connect to the IOs respectively.<br><br>The operator is in prototype library and might be replaced in future releases. | |

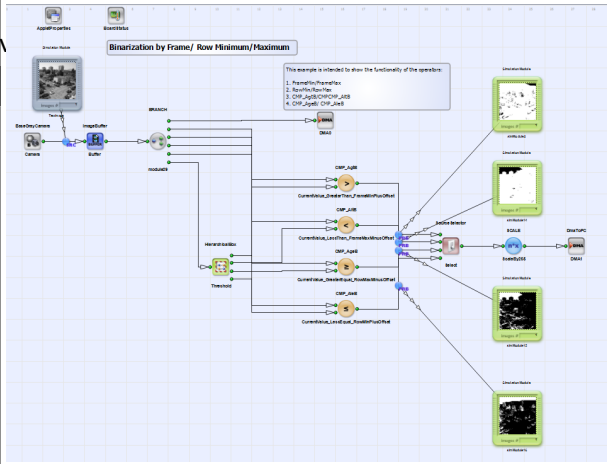
# 13. Operator Examples

The VisualApplets example designs in this chapter demonstrate how to use and parameterize specific operators. All examples can be found in the VisualApplets installation directory in sub-directory examples i.e. "**%VASINSTALLDIR%/examples/Examples/OperatorExamples**".

A detailed introduction into the libraries you find in the *Operator Reference*.

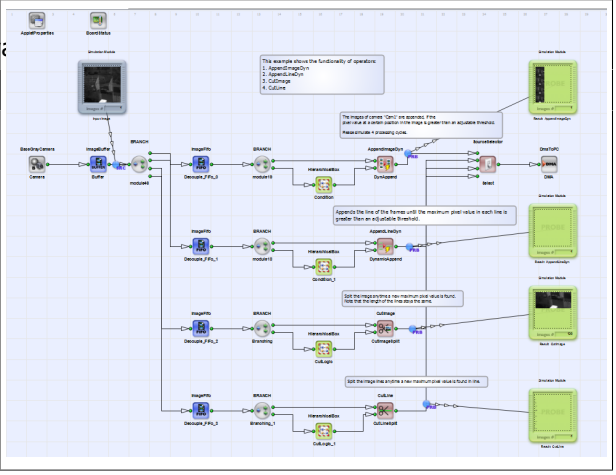
The following sections show an overview on all examples. A brief introduction is presented as well as the folder and filename. All examples use a default hardware platform. Nearly most of them can easily be converted to other hardware platforms. See Section 4.5, 'Target Hardware Porting' for more information on how to switch the hardware platform.

## 13.1. Functional Example for Specific Operators of Library Accumulator and Library Logic

| **Brief Description** | |
|---|---|
| **Files:** examples\OperatorExamples \AccumulatorLibrary_LogicalLibrary_Thresholding.\ |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Demonstration of how to use the operators:<br>• 1. *FrameMin* (library:accumulator)<br>• 2. *FrameMax* (library:accumulator)<br>• 3. *RowMin* (library:accumulator)<br>• 4. *RowMax* (library:accumulator)<br>• 5. *CMP_AgtB* (library:logic)<br>• 6. *CMP_AltB* (library:logic)<br>• 7. *CMP_AgeB* (library:logic)<br>• 8. *CMP_AleB* (library:logic) | |

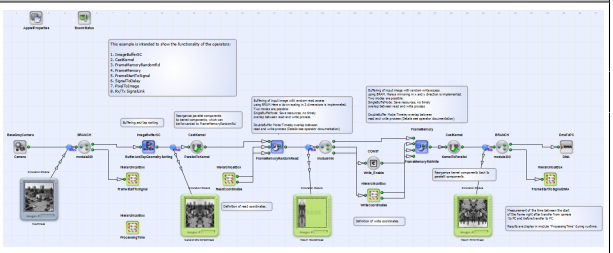
This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the accumulator and signal library in the Operator Reference, section 18. *Library Accumulator* [592] and 26. *Library Logic* [909].

## 13.2. Functional Example for Specific Operators of Library Synchronization: Dynamic Append and Cut

| Brief Description | |
|---|---|
| **Files:** examples\OperatorExamples \SynchronizationLibrary_DynamicAppendAndCut.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Demonstration of how to use the operators:<br>• 1. *AppendImageDyn* (library:synchronization)<br>• 2. *AppendLineDyn* (library:synchronization)<br>• 3. *CutImage* (library:synchronization)<br>• 4. *CutLine* (library:synchronization) |  |

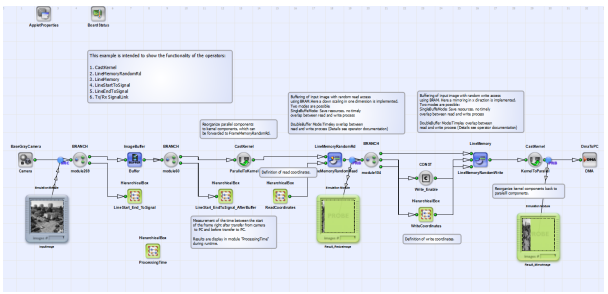
This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the arithmetics library in the Operator Reference, section 32. *Library Synchronization* [1491].

## 13.3. Functional Example for Specific Operators of Library Memory and Library Signal

| Brief Description | |
|---|---|
| **Files:** examples\OperatorExamples \MemoryLibrary_Frame_LatencyMeasurement.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Demonstration of how to use the operators:<br>• 1. *CastKernel* (library: base)<br>• 2. *FrameMemoryRandomRd* (library: memory)<br>• 3. *FrameMemory* (library: memory)<br>• 4. *ImageBufferSC* (library: memory)<br>• 5. *FrameStartToSignal* (library: signal)<br>• 6. *SignalToDelay* (library: signal)<br>• 7. *TxSignalLink* (library: signal)<br>• 8. *RxSignalLink* (library: signal)<br>• 8. *PixelToImage* (library: synchronization) |  |

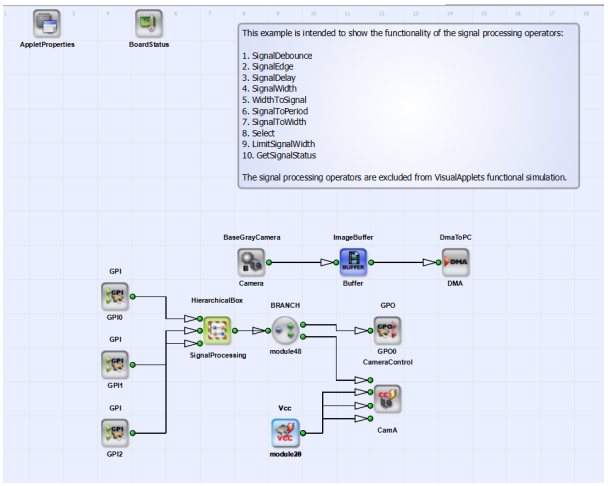
This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the specific library in the Operator Reference, section 27. *Library Memory* [951], 31. *Library Signal* [1396] and 32. *Library Synchronization* [1491].

## 13.4. Functional Example for Specific Operators of Library Memory and Library Signal

| Brief Description | |
|---|---|
| **Files:** examples\OperatorExamples \MemoryLibrary_Line_LatencyMeasurement.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Demonstration of how to use the operators:<br>• 1. *CastKernel* (library: base)<br>• 2. *LineMemoryRandomRd* (library: memory)<br>• 3. *LineMemory* (library: memory)<br>• 4. *LineStartToSignal* (library: signal)<br>• 5. *LineEndToSignal* (library: signal)<br>• 6. *TxSignalLink* (library: signal)<br>• 7. *RxSignalLink* (library: signal) |  |

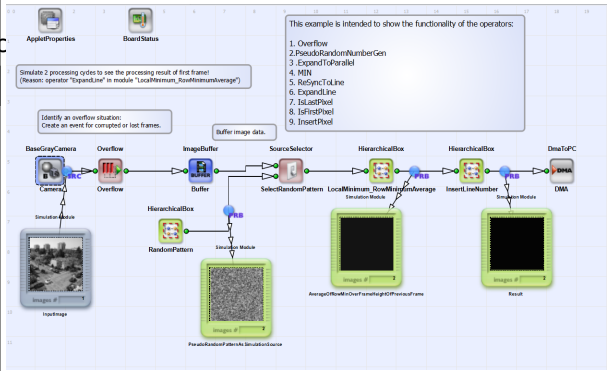
This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the memory and signal library in the Operator Reference, section 27. *Library Memory* [951] and 31. *Library Signal* [1396].

## 13.5. Functional Example for Specific Operators of Library Signal

| Brief Description | |
|---|---|
| **Files:** examples\OperatorExamples \SignalLibrary_SignalProcessing.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Demonstration of how to use the operators:<br>• 1. *SignalDebounce* (library:signal)<br>• 2. *SignalEdge* (library:signal)<br>• 3. *SignalDelay* (library:signal)<br>• 4. *SignalWidth* (library:signal)<br>• 5. *WidthToSignal* (library:signal)<br>• 6. *SignalToPeriod* (library:signal)<br>• 7. *SignalToWidth* (library:signal)<br>• 8. *Select* (library:signal)<br>• 9. *LimitSignalWidth* (library:signal)<br>• 10. *GetSignalStatus* (library:signal) |  |

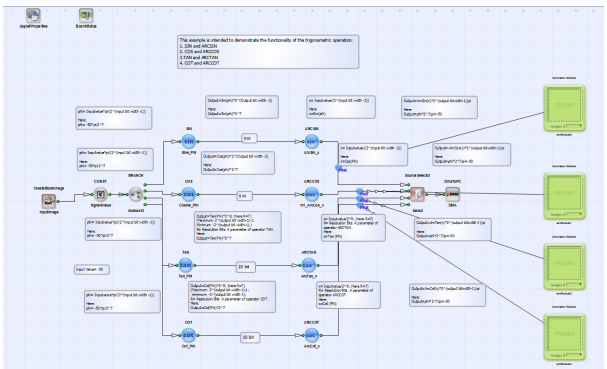
This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the signal library in the Operator Reference, section 31. *Library Signal* [1396].

## 13.6. Functional Example for Specific Operators of Library Synchronization, Base and Filter

| Brief Description | |
|---|---|
| **Files:** examples\OperatorExamples \SynchronizationLibrary_LocalAndGlobalImageProc |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Demonstration of how to use the operators:<br>• 1. *Overflow* (library:synchronization)<br>• 2. *PseudoRandomNumberGen* (library: base)<br>• 3. *ExpandToParallel* (library:base)<br>• 4. *MIN* (library:filter)<br>• 5. *ReSyncToLine* (library:synchronization)<br>• 6. *ExpandLine* (library:synchronization)<br>• 7. *IsLastPixel* (library:synchronization)<br>• 8. *IsFirstPixel* (library:synchronization)<br>• 9. *InsertPixel* (library:synchronization) | |

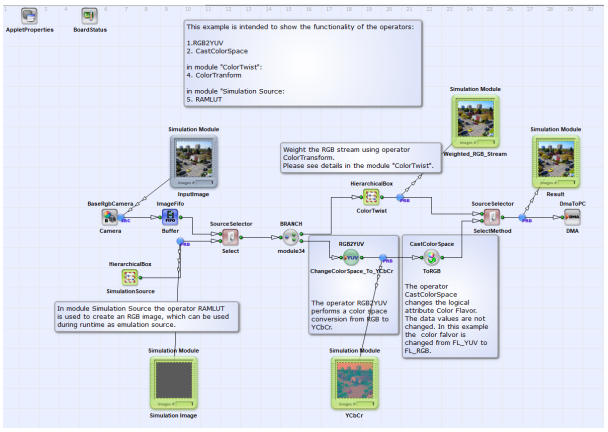
This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the operators of the specific library in the Operator Reference, section 32. *Library Synchronization* [1491], 20. *Library Base* [674], 25. *Library Filter* [882] .

## 13.7. Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions

| Brief Description | |
|---|---|
| **Files:** examples\OperatorExamples \ArithmeticsLibrary_TrigonometricFunctions.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Demonstration of how to use the operators:<br>• 1. *SIN* (library:arithmetics)<br>• 2. *ARCSIN* (library:arithmetics)<br>• 3. *COS* (library:arithmetics)<br>• 4. *ARCCOS* (library:arithmetics)<br>• 5. *TAN* (library:arithmetics)<br>• 6. *ARCTAN* (library:arithmetics)<br>• 7. *COT* (library:arithmetics)<br>• 8. *ARCCOT* (library:arithmetics) | |

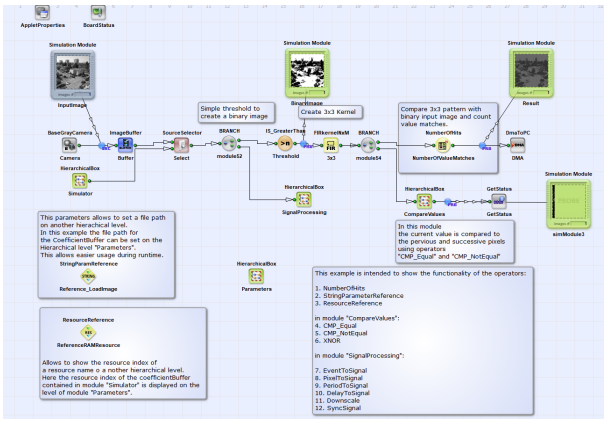
This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the arithmetics library in the Operator Reference, section 19. *Library Arithmetics* [623].

## 13.8. Functional Example for Specific Operators of Library Color, Base and Memory

| **Brief Description** | |
|---|---|
| **Files:** examples\OperatorExamples \ColorLibrary_ColorTransformation.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Demonstration of how to use the operators:<br>• 1. *RGB2YUV* (library:color)<br>• 2. *ColorTransform* (library:color)<br>• 3. *CastColorSpace* (library:base)<br>• 4. *RamLUT* (library:memory) |  |

This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the operators of the specific library in the Operator Reference, section 22. *Library Color* [799], 20. *Library Base* [674] and 27. *Library Memory* [951].

## 13.9. Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters

| **Brief Description** | |
|---|---|
| **Files:** examples\OperatorExamples \SignalLibrary_ImageStreamToSignal.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Demonstration of how to use the operators:<br>• 1. *NumberOfHits* (library:filter)<br>• 2. *StringParamReference* (library:parameters)<br>• 3. *ResourceReference* (library:parameters)<br>• 4. *CMP_Equal* (library:logic)<br>• 5. *CMP_NotEqual* (library:logic)<br>• 6. *XNOR* (library:logic)<br>• 7. *EventToSignal* (library:signal)<br>• 8. *PixelToSignal* (library:signal)<br>• 9. *PeriodToSignal* (library:signal)<br>• 10. *DelayToSignal* (library:signal) |  |

| Brief Description | |
|---|---|
| • 11. *Downscale* (library:signal)<br>• 12. *SyncSignal* (library:signal) | |

This example demonstrates how the specific operators listed above can be used in a functional design. You find a detailed introduction to the operators of the specific library in the Operator Reference, section 31. *Library Signal* [1396], 26. *Library Logic* [909], 25. *Library Filter* [882] and 28. *Library Parameters* [1049].

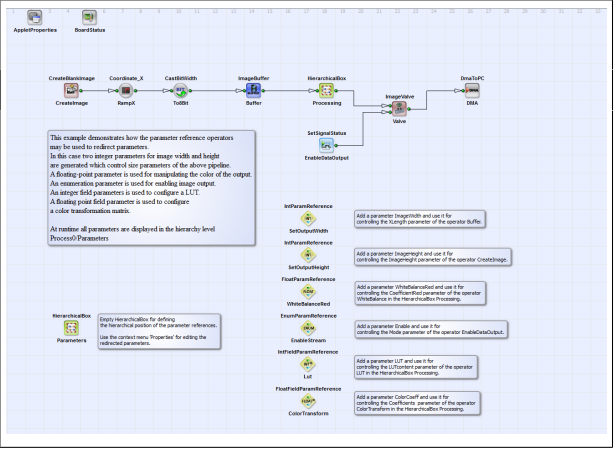
# 14. Parameter Library Examples

The VisualApplets example designs in this chapter demonstrate how to use and manipulate parameters. All examples can be found in the VisualApplets installation directory in sub-directory examples i.e. "**%VASINSTALLDIR%/examples/AdvancedVAFunctions/Parameters Library**".

A detailed introduction into library Parameters you find in the *Operator Reference*, section *Library Parameters*.

The following sections show an overview on all examples. A brief introduction is presented as well as the folder and filename. All examples use a default hardware platform. Nearly most of them can easily be converted to other hardware platforms. See Section 4.5, 'Target Hardware Porting' for more information on how to switch the hardware platform.

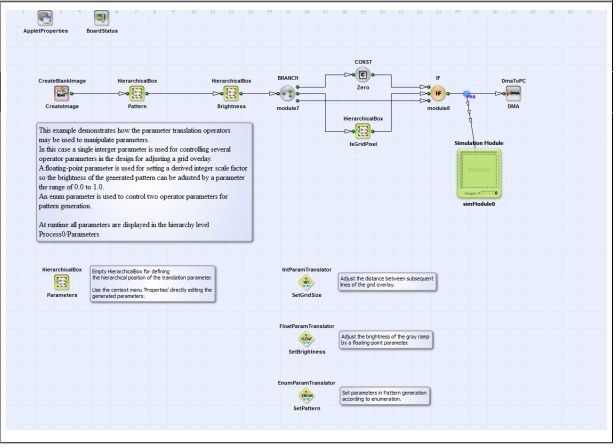
## 14.1. Parameter Redirection

| **Brief Description** | |
|---|---|
| **Files:** \examples\AdvancedVAFunctions \Parameters Library\ParameterRedirection.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>Demonstration how to use the parameter reference operators. | |

This example demonstrates how the parameter reference operators may be used to redirect parameters. In this case two integer parameters for image width and height are generated which control size parameters of the above pipeline. A floating-point parameter is used for manipulating the color of the output. An enumeration parameter is used for enabling image output. An integer field parameters is used to configure a LUT. A floating point field parameter is used to configure a color transformation matrix. At runtime all parameters are displayed in the hierarchy level Process0/Parameters.

You find a detailed introduction to the Parameters Library in the Operator Reference, section 28. *Library Parameters* [1049].

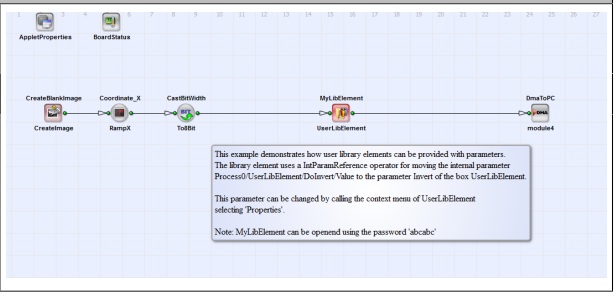
## 14.2. Parameter Translation

| Brief Description | |
|---|---|
| **Files:** \examples\AdavancedVAFunctions \Parameters Library\ParameterTranslation.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Demonstration how to use the parameter translation operators for manipulation of parameters. |  |

This example demonstrates how the parameter translation operators may be used to manipulate parameters. In this case a single interger parameter is used for controlling several operator parameters in the design for adjusting a grid overlay. A floating-point parameter is used for setting a derived integer scale factor so the brightness of the generated pattern can be adusted by a parameter the range of 0.0 to 1.0. An enum parameter is used to control two operator parameters for pattern generation. At runtime all parameters are displayed in the hierarchy level Process0/Parameters.

You find a detailed introduction to the Parameters Library in the Operator Reference, section 28. *Library Parameters* [1049].

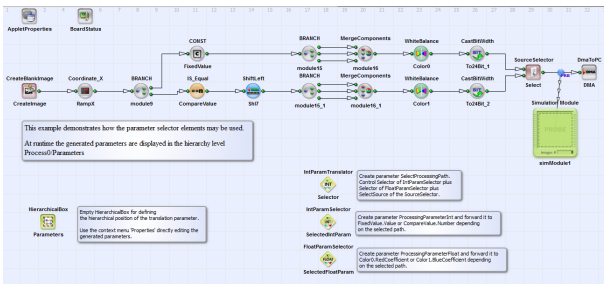
## 14.3. User Library Parameter

| Brief Description | |
|---|---|
| **Files:** \examples\AdavancedVAFunctions \Parameters Library\UserLibParameter.va<br>**Default Platform:** mE5-MA-VCL<br>**Short Description**<br>Demonstration how user library elements can be provided with parameters. |  |

This example demonstrates how user library elements can be provided with parameters. The library element uses a IntParamReference operator for moving the internal parameter Process0/UserLibElement/DoInvert/Value to the parameter Invert of the box UserLibElement. This parameter can be changed by calling the context menu of UserLibElement selecting 'Properties'. Note: MyLibElement can be openend using the password 'abcabc'.

You find a detailed introduction to the Parameters Library in the Operator Reference, section 28. *Library Parameters* [1049].

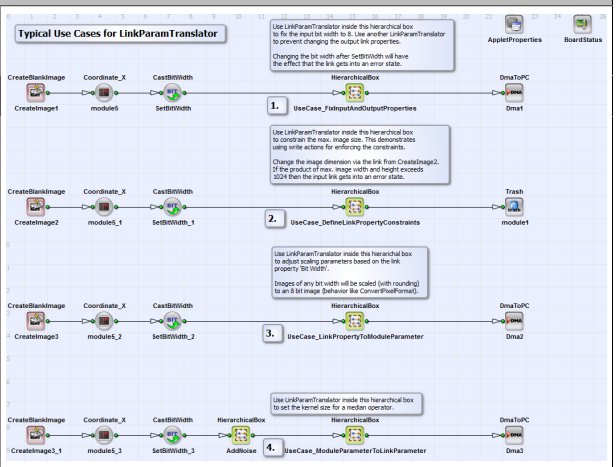
## 14.4. Parameter Selection

| Brief Description | |
|---|---|
| **Files:** \examples\AdavancedVAFunctions \Parameters Library\ParameterSelection.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A demonstration of how to use the parameter translation operators *IntParamSelector* and *FloatParamSelector*. | |

This example demonstrates how the operators *IntParamSelector* and *FloatParamSelector* may be applied for redirecting parameter access to different module parameters depending on a selector.

You find a detailed introduction to the Parameters Library in the Operator Reference, section 28. *Library Parameters* [1049].

## 14.5. Link Parameter Translation

| Brief Description | |
|---|---|
| **Files:** \examples\AdavancedVAFunctions \Parameters Library\LinkParamTranslator.va |  |
| **Default Platform:** mE5-MA-VCL | |
| **Short Description**<br><br>A demonstration of how to use the parameter translation operator *LinkParamTranslator*. | |

This example demonstrates how the operator *LinkParamTranslator* may be applied for different use cases:

- Fixing input and output link properties for a hierarchical box
- Defining additional constraints on link properties
- Translating link properties to module parameters
- Translating module parameters to link properties

You find a detailed introduction to the Parameters Library in the Operator Reference, section 28. *Library Parameters* [1049].

# 15. Using Applets During Runtime

In this chapter you find examples for using applets during runtime.

## 15.1. Filling LUT with Content With the Basler Framegrabber API

In this section you find an example how you can easily fill a LUT with content during runtime. The method is based on the struct *FieldParameterAccess*. Documentation for the struct FieldParameterAccess [https://docs.baslerweb.com/frame-grabbers/sdk/struct_field_parameter_access.html] and the function Fg_setParameterWithType() [https://docs.baslerweb.com/frame-grabbers/sdk/basler__fg_8h.html] is available in the Basler Framegrabber API documentation.

**Example Implementation:**

This struct *FieldParameterAccess* allows access to array parameters of any type in a flexible way. Range accesses as well as single value accesses are both possible.

**Example:**

```
# include <fgrab_struct.h>

struct FieldParameterAccess singleaccess;

struct FieldParameterAccess rangeaccess;
```

To fill the LUT with content with the Basler Framegrabber API:

1. **Define the values of the LUT content**

   Example:

   ```
   uint64_t primes[7] = { 1, 2, 3, 5, 7, 11, 13 };

   uint32_t answer = 42;
   ```

2. **Set up single value access or range access:**

   1. FieldParameterAccess:: vtype: set the type of the included data
   2. FieldParameterAccess:: index: set the first index in the range
   3. FieldParameterAccess:: count: set the value count of the range
   4. A range of values according to the type of the included data: FieldParameterAccess:: p_double, p_int32_t, p_int64_t, p_uint32_t, or puint64_t

   Example:

   ```
   // set up single value access

   singleaccess.vtype = FG_PARAM_TYPE_UINT32_T;

   singleaccess.index = 17;

   singleaccess.count = 1;
   ```

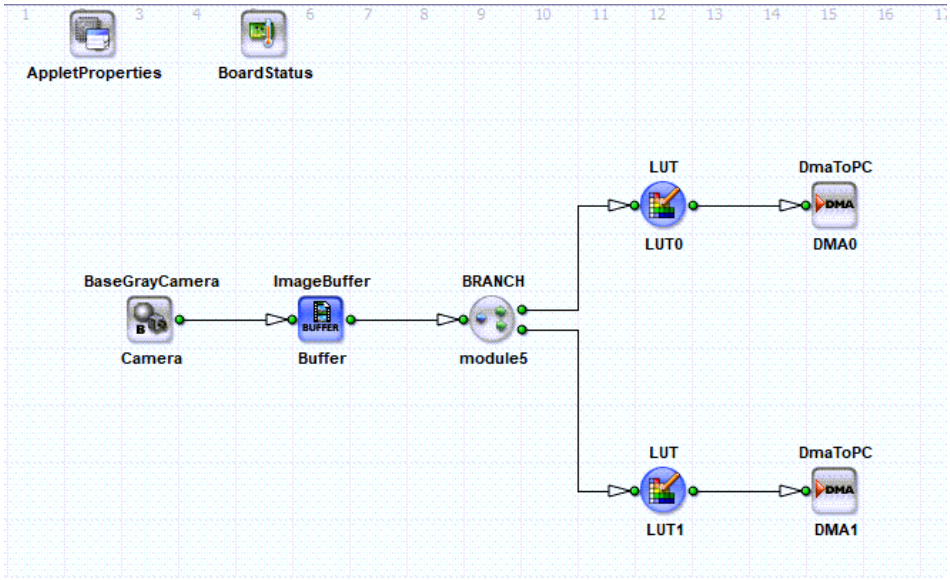
```
singleaccess.p_uint32_t = &answer;


// set up range access
rangeaccess.vtype = FG_PARAM_TYPE_UINT64_T// every data value is an uint64_t
rangeaccess.index = 0;
rangeaccess.count = 7;
rangeaccess.p_uint64_t = primes;
```

3. **Write the content into the applet operator LUT**

   Write the content into the applet Operator LUT [https://docs.baslerweb.com/visualapplets/files/manuals/content/Memory.LUT.html] using the function Fg_setParameterWithType() [https://docs.baslerweb.com/frame-grabbers/sdk/basler__fg_8h.htm].

   Example:




4. **Write the LUT Content to the Corresponding LUT Operators in the Compiled Hardware Applet**

   For a VisualApplets design structure with two LUT operators as shown in screenshot above, you can write the LUT content to the corresponding LUT operators in the compiled hardware applet as:

```
retCode += Fg_setParameterWithType(fg, Fg_getParameterIdByName(fg, "Device1_Process0_LUT0 _LUTcontent"),
&singleaccess, 0, FG_PARAM_TYPE_STRUCT_FIELDPARAMACCESS);

retCode += Fg_setParameterWithType(fg, Fg_getParameterIdByName(fg, "Device1_Process0_LUT1 _LUTcontent"),
&rangeaccess, 0, FG_PARAM_TYPE_STRUCT_FIELDPARAMACCESS);
```

# Part III

## Operator Reference

# 16. Introduction

VisualApplets comprises numerous operators which represent processing functionalities. This part explains the functionality of all available operators in detail. The operator reference is structured as explained in the following:

- **Description**

  In the description the main functionality and behavior of the operator is explained. It is the main part of the operator reference.

  For operators of the library *Hardware Platform*, a table is included which shows on which hardware platforms the respective operator is available.

- **Operator Restrictions**

  If the operator has restrictions in input format or simulation, these issues are listed here.

- **I/O Properties**

  Shows a table where

  - The operator type is given. The type can either be O-, M- or P-type. (see Section 3.6.1, 'Operator Types')
  - The input links are listed.
  - The output links are listed.

  If the operator is of type M and has more than one input, synchronous and asynchronous input groups are listed. See Section 3.6.4, 'M-type Operators with Multiple Inputs' for more information on input groups.

- **Supported Link Formats**

  The supported link formats for all input and output ports are listed. See Section 3.7.2, 'Link Properties' for an explanation of all link properties.

- **Parameters**

  If the operator has parameters, they are listed in this section. For each parameter besides its description, the name, the value range and the default value is given. To learn more about parameterization, check Section 3.7.1, 'Module Properties'.

- **Examples of Use**

  The examples of use section is a very helpful part. It shows a list of examples where the operator is used. You can directly click on a link to get redirected to the example. The example of use can be anywhere in the user manual or tutorial and examples parts.

- **More Information**

  Some operators have more detailed information at the end of the operator reference.

The operator reference can directly be accessed from VisualApplets. See Section 3.3.1.6, 'Help' for more information. Moreover, the Index is a good resource to quickly access the required operator references. Finally, in 17. *Library Overview* a list of all available libraries is provided.

# 17. Library Overview

| Library Name | | Short Description |
|---|---|---|
|  | Accumulator | Accumulation operators such as counters. |
|  | Arithmetics | Arithmetic operators such as ADD, SUB, COS, ShiftLeft, ... |
|  | Base | Basic operators for common functions. |
|  | Blob | Blob Analysis operators. |
|  | Color | Includes operators for color space transformations and color processing. |
|  | Compression | Compression operators such as JPEG. |
|  | Debugging | Debugging operators. |
|  | Filter | Includes filter operators for any kind of image filters and kernel operations. |
|  | Logic | Includes operators for logic operations such as comparisons and Boolean operations. |
|  | Memory | Operators which can store data in memory. |
|  | Parameters | Contains Operators for translating parameter values. |
|  | Hardware Platform | This library contains operators which are only available on certain platforms. |
|  | Prototype | Prototype operators. |
|  | Signal | Operators for signal data processing such as trigger signals. |

| Library Name | | Short Description |
|---|---|---|
|  | Synchronization | Includes operators for image synchronizations such as removing, appending, splitting, inserting, ... of images, lines and pixels. |
|  | Transformation | Transformation operators. |

Table 17.1. Available Libraries

# 18. Library Accumulator



Operators of library *Accumulator* include accumulating operators such as counters.

The following list summarizes all Operators of Library Accumulator

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | ColMax | Returns the maximum pixel value for each image column. | Version 1.2 |
|  | ColMin | Returns the minimum pixel value for each image column. | Version 1.2 |
|  | ColSum | Computes the sum of all pixel values for each image column. | Version 1.2 |
|  | Count | Up and down counter with reset. | Version 1.2 |
|  | FrameMax | Returns the maximum pixel value for each frame. | Version 1.2 |
|  | FrameMin | Returns the minimum pixel value for each frame. | Version 1.2 |
|  | FrameSum | Computes the sum of all pixel values for each frame. | Version 1.2 |
|  | Histogram | Computes an intensity histogram of the input image. | Version 1.3 |
|  | ModuloCount | Counts the number of pixels, lines or frames at the input link. | Version 1.3 |
|  | Register | A register with data and capture input. | Version 1.3 |
|  | RowMax | Returns the maximum pixel value for each image row. | Version 1.2 |
|  | RowMin | Returns the minimum pixel value for each image row. | Version 1.2 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | RowSum | Computes the sum of the pixel values for each image row. | Version 1.2 |

Table 18.1. Operators of Library Accumulator

# 18.1. Operator ColMax

**Operator Library: Accumulator**

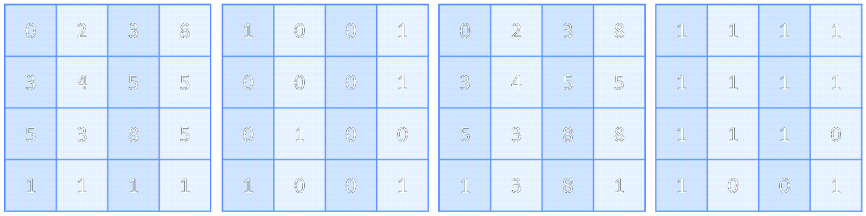
This operator returns the maximum pixel value found inside each image column. The operator replaces each pixel value by the maximum value found in the very same column so far. Only if a current pixel is a new maximum, the output *IsMaxO* is set to 1. Otherwise, it is 0.

It is possible to clear the current maximum manually by setting the *ClrI* link to 1. When a manual clear is not required, a constant 0 must be applied to *ClrI*.

With every new input frame, the detection of the maximum value restarts for each column.

Often, this operator is used in conjunction with the *RemoveLine* operator to remove all but the last line of a frame. This last line of a frame contains the results of the *ColMax* operator.

The following example shows the relation of input values to output values:




**Operator Restrictions:**

- Image Size

  All lines of an input image must have the same length. Images with varying line lengths are not allowed.

- Empty Images

  Empty Images, i.e., images with no pixels, are allowed. The operator will output an empty image.

- Image Protocols

  Only VALT_IMAGE2D and VALT_LINE1D image protocols are supported.

## 18.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Links** | O, Data Output<br>IsMaxO, Binary output to indicate a new maximum |

## 18.1.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D,VALT_LINE1D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O | Output Link IsMaxO |
|---|---|---|
| **Bit Width** | as I | 1 |
| **Arithmetic** | as I | unsigned |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | as I |
| **Color Flavor** | as I | as I |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

### 18.1.3. Parameters

None

### 18.1.4. Examples of Use

The use of operator ColMax is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

# 18.2. Operator ColMin

**Operator Library: Accumulator**

This operator returns the minimum pixel value found for each image column. The operator replaces each pixel value by the minimum value found in the very same column so far. If a current pixel is a new minimum, the output link *IsMinO* is set to 1, otherwise it is 0.

It is possible to clear the current minimum manually by setting the *ClrI* link to 1. When a manual clear is not required, a constant 0 must be applied to *ClrI*.

With every new input frame, the detection of the minimum value restarts for each column.

Often, this operator is used in conjunction with the *RemoveLine* operator to remove all but the last line of a frame. This last line of a frame contains the results of the *ColMin* operator.

The following example shows the relation of input values to output values:

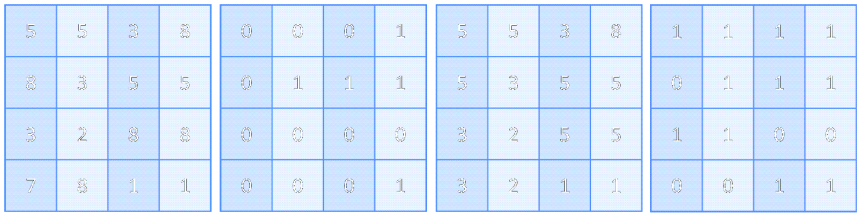


**Image Restrictions:**

- Image Size

  All lines of an input image must have the same length. Images with varying line lengths are not allowed.

- Empty Images

  Empty Images, i.e., images with no pixels, are allowed. The operator will output an empty image.

- Image Protocols

  Only VALT_IMAGE2D and VALT_LINE1D image protocols are supported.

## 18.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Links** | O, Data Output<br>IsMinO, Binary output to indicate a new minimum |

## 18.2.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D,VALT_LINE1D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O | Output Link IsMinO |
|---|---|---|
| **Bit Width** | as I | 1 |
| **Arithmetic** | as I | unsigned |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | as I |
| **Color Flavor** | as I | as I |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

### 18.2.3. Parameters

None

### 18.2.4. Examples of Use

The use of operator ColMin is shown in the following examples:

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

# 18.3. Operator ColSum

**Operator Library: Accumulator**

This operator calculates the sum of all pixel values for each image column. Each pixel value at the output represents the current sum of the respective column. Thus, each pixel is replaced by the sum determined so far.

It is possible to clear the current column sum manually by setting the *ClrI* link to 1. When a manual clear is not required, a constant 0 must be applied to *ClrI*.

With every new input frame, the sum for all columns is set to zero.

Often, this operator is used in conjunction with operator *RemoveLine* to use only the last image line containing the results for the full frame.

The following example shows the relation of input values to output values:

Frame I

| | | | |
|---|---|---|---|
| 0 | 2 | 3 | 8 |
| 3 | 4 | 5 | 5 |
| 5 | 3 | 8 | 5 |
| 1 | 1 | 1 | 1 |

Frame ClrI

| | | | |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |

Frame O

| | | | |
|---|---|---|---|
| 0 | 2 | 3 | 8 |
| 3 | 6 | 8 | 5 |
| 8 | 9 | 16 | 10 |
| 1 | 10 | 17 | 1 |

**Image Restrictions:**

- Image Size

  All lines of an input image must have the same length. Images with varying line lengths are not allowed.

- Empty Images

  Empty Images, i.e,. images with no pixels, are allowed. The operator will output an empty image.

- Image Protocols

  Only VALT_IMAGE2D and VALT_LINE1D image protocols are supported.

## 18.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Link** | O, Data Output |

## 18.3.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 | auto/manual❶❷ |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |

| Link Parameter | Input Link I | Input Link ClrI | Output Link O |
|---|---|---|---|
| **Kernel Columns** | 1 | as I | as I |
| **Kernel Rows** | 1 | as I | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I | as I |
| **Color Format** | VAF_GRAY | as I | as I |
| **Color Flavor** | FL_NONE | as I | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ For the image protocol VALT_IMAGE2D, the output bit width is automatically calculated from the input bit width and the maximum image height. The output bit width is determined by

$$OutputBitWidth = InputBitWidth \times \lceil log_2(Max.ImgHeight + 1) \rceil$$

The output bit width must not exceed 64 Bit.

❷ For the image protocol VALT_LINE1D, the output bit width must be set manually on the output link.

The output bit width must be greater than 2 and must not exceed 64 bit.

### 18.3.3. Parameters

None

### 18.3.4. Examples of Use

The use of operator ColSum is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

# 18.4. Operator Count

**Operator Library: Accumulator**

This up and down counter counts all pixels with value "1" at input I. It accepts only 1-bit pixel input. It is possible to define an initialization value, a direction (count up/count down), and an automatic clear strategy (end-of-line/end-of-frame reset) as parameters. Additionally, it is possible to reset the counter manually through the *ClrI* input link.

Depending on the *AutoClear* strategy, the output bit width is determined by the image width or image size.

The output frame has the same size as the input frame. For every input pixel, the operator will output the current counter value. The counter wraps around if the counter reaches maximum in count-up mode or zero in count-down mode.

The following example shows the relation of input values to output values. In this example the *AutoClear* strategy is set to EoL (end-of-line). The initialization value *Init* is "0".

Frame I

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |

Frame ClrI

| | | | |
|---|---|---|---|
| 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |

Frame O

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 1 |

This operator is commonly used with Logic operators.

**Operator Restrictions**

- Empty Images

  Empty Images, i.e. images with no pixels are allowed. The operator will output an empty image.

- Image Size

  All lines of each input image may have varying lengths but must not be empty.

## 18.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Link** | O, Data Output |

## 18.4.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI | Output Link O |
|---|---|---|---|
| **Bit Width** | 1 | 1 | auto❶ |
| **Arithmetic** | unsigned | as I | unsigned |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | 1 | as I | as I |
| **Kernel Rows** | 1 | as I | as I |

| Link Parameter | Input Link I | Input Link ClrI | Output Link O |
|---|---|---|---|
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I | as I |
| **Color Format** | VAF_GRAY | as I | as I |
| **Color Flavor** | FL_NONE | as I | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The output bit width is automatically determined from the input image dimensions and the parameter settings. Output bit width for image protocol VALT_IMAGE2D and parameter *AutoClear* = EoL or image protocol VALT_LINE1D:

$$OutputBitWidth = \lceil log_2(Max.ImgWidth + 1) \rceil$$

Output bit width for image protocol VALT_IMAGE2D and parameter *AutoClear* = EoF or image protocol VALT_PIXEL0D:

$$OutputBitWidth = \lceil log_2(Max.ImgWidth \times Max.ImgHeight + 1) \rceil$$

When image protocol VALT_PIXEL0D is used or *AutoClear* is set to *None*, the output bit width can be adjusted directly at the output link *O*.

The output bit width must not exceed 64 Bit.

## 18.4.3. Parameters

| **AutoClear** | |
|---|---|
| **Type** | static parameter |
| **Default** | EoL |
| **Range** | {EoL, EoF, None} |
| This parameter defines when to reset the counter to its initial value define by parameter Init.<br><br>The parameter cannot be set to EoF if image protocol VALT_LINE1D is used. It is disabled for image protocol VALT_PIXEL0D. | |

| **Init** | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 2^OutputBitWidth - 1] |
| This parameter defines the initial value. The counter is set to its initial value at<br><br>• start-up<br><br>• CltI = 1<br><br>• AutoClear condition | |

| **Direction** | |
|---|---|
| **Type** | static parameter |
| **Default** | UP |
| **Range** | {UP, DOWN} |
| This parameter defines whether the counter is incremented (UP) or decremented (DOWN) for each value '1' at the input link I. | |

### 18.4.4. Examples of Use

The use of operator Count is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

## 18.5. Operator FrameMax

**Operator Library: Accumulator**

This operator returns the maximum pixel value found inside each input frame. The output frame has the same size as the input frame, where each pixel is replaced by the maximum value found in the image so far. If the current pixel is a new maximum the output Link *IsMaxO* is 1, otherwise it is 0.



**Resource Consumption**

The operator's FGPA resource consumption strongly increases with the parallelism. We recommend to use low parallelism at this operator.

In many use cases of this operator, the parallelism can be reduced selecting the max parallel component prior to this operator. Use *SplitParallel*, *MergeKernel* and *MAX* for this operation.

With every new input frame the maximum detection starts again. Additionally, it is possible to clear the current maximum manually when the *ClrI* link is set to 1. When a manual clear is not necessary, you must ensure a constant 0 at *ClrI*, to enable normal operation.

The following example shows the relation of the input pixel to the output values.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *I* | 10 | 12 | 9 | 5 | 6 | 4 | 2 |
| *ClrI* | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| *MaxO* | 10 | 12 | 12 | 12 | 6 | 6 | 2 |
| *IsMaxO* | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

Often, this operator is used in conjunction with operators *RemoveLine* and *RemovePixel* to use only the last image pixel containing the results of all frame pixels.

**Image Restrictions**

- Empty Images

  Empty images, i.e., images with no pixels, are allowed. The operator will output an empty image.

### 18.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Links** | O, Data Output<br>IsMaxO, binary output to indicate a new maximum |

### 18.5.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D, VALT_IMAGE1D, VALT_IMAGE0D | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O | Output Link IsMaxO |
|---|---|---|
| **Bit Width** | as I | 1 |
| **Arithmetic** | as I | unsigned |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | as I |
| **Color Flavor** | as I | as I |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

### 18.5.3. Parameters

None

### 18.5.4. Examples of Use

The use of operator FrameMax is shown in the following examples:

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 13.1, 'Functional Example for Specific Operators of Library Accumulator and Library Logic'

  Examples - Demonstration of how to use the operator

# 18.6. Operator FrameMin

**Operator Library: Accumulator**

This operator returns the minimum pixel value found for each input frame. The output frame has the same size as the input frame, where each pixel is replaced by the minimum value found in the image so far. If the current pixel is a new minimum the output Link *IsMinO*, otherwise it is 0.



**Resource Consumption**

The operator's FGPA resource consumption strongly increases with the parallelism. We recommend to use low parallelism at this operator.

In many use cases of this operator, the parallelism can be reduced selecting the min parallel component prior to this operator. Use *SplitParallel*, *MergeKernel* and *MIN* for this operation.

With every new input frame the minimum detection starts again. Additionally, it is possible to clear the current minimum manually when the *ClrI* link is set to 1. When a manual clear is not necessary, you must ensure a constant 0 at *ClrI*, to enable normal operation.

The following example shows the relation of input values to output values.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *I* | 10 | 12 | 9 | 5 | 6 | 4 | 2 |
| *ClrI* | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| *MinO* | 10 | 10 | 9 | 5 | 6 | 4 | 2 |
| *IsMinO* | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

Often, this operator is used in conjunction with operators *RemoveLine* and *RemovePixel* to use only the last image pixel containing the results of all frame pixels.

**Image Restrictions**

- Empty Images

  Empty images, i.e., images with no pixels, are allowed. The operator will output an empty image.

## 18.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Links** | O, Data Output<br>IsMinO, binary output to indicate a new minimum |

## 18.6.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D, VALT_IMAGE1D, VALT_IMAGE0D | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O | Output Link IsMinO |
|---|---|---|
| **Bit Width** | as I | 1 |
| **Arithmetic** | as I | unsigned |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | as I |
| **Color Flavor** | as I | as I |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

### 18.6.3. Parameters

None

### 18.6.4. Examples of Use

The use of operator FrameMin is shown in the following examples:

- Section 13.1, 'Functional Example for Specific Operators of Library Accumulator and Library Logic'

  Examples - Demonstration of how to use the operator

# 18.7. Operator FrameSum

**Operator Library: Accumulator**

This operator computes the sum of all pixel values for each input frame. The output frame has the same size as the input frame, where each pixel is replaced by the current frame sum. Thus, each pixel is replaced by the pixel sum determined so far.



**Resource Consumption**

The operator's FGPA resource consumption strongly increases with the parallelism. We recommend to use low parallelism at this operator.

In many use cases of this operator, the parallelism can be reduced by adding up the parallel components prior this operator using *SplitParallel* and *ADD*.

With every new input frame the summation starts from zero. Additionally, it is possible to clear the current sum manually when the *ClrI* link is set to 1. When a manual clearing is not necessary, you must ensure a constant 0 at *ClrI*, to enable normal operation.

The following example shows the relation of input values to output values.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *I* | 10 | 12 | 9 | 5 | 6 | 4 | 2 |
| *ClrI* | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| *O* | 10 | 22 | 31 | 36 | 6 | 10 | 2 |

Often, this operator is used in conjunction with operators *RemoveLine* and *RemovePixel* to use only the last image pixel containing the results of all frame pixels.

**Image Restrictions**

- Image size

  Varying line lengths and empty lines are allowed in non-empty images.

- Empty Images

  Empty images, i.e., images with no pixels, are allowed. The operator will output an empty image.

## 18.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Link** | O, Data Output |

## 18.7.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 | auto❶ |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |

| Link Parameter | Input Link I | Input Link ClrI | Output Link O |
|---|---|---|---|
| **Kernel Columns** | 1 | as I | as I |
| **Kernel Rows** | 1 | as I | as I |
| **Img Protocol** | VALT_IMAGE2D | as I | as I |
| **Color Format** | VAF_GRAY | as I | as I |
| **Color Flavor** | FL_NONE | as I | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The output bit width is automatically determined from the input bit width and maximum image dimension. The output bit width is determined by

$$OutputBitWidth = InputBitWidth + \lceil log_2(Max.ImgWidth * Max.ImgHeight + 1) \rceil$$

The output bit width must not exceed 64 Bit.

## 18.7.3. Parameters

None

## 18.7.4. Examples of Use

The use of operator FrameSum is shown in the following examples:

- Section 12.2.2, 'Auto Threshold Mean'

  Determines the mean value of an image and used the value as threshold value for the next image processed.

# 18.8. Operator Histogram

**Operator Library: Accumulator**

This operator calculates an intensity histogram from the input image at *I* and will output the results at output *O*. Depending on the option of the *AutoSync* parameter, the histogram is computed for each line or for each frame. The output is not a visualization of the results, but a list of the histogram values.

- AutoSync = EoF produces a histogram image of height 1 for the entire frame.
- AutoSync = EoL produces a histogram image of the input image height, i.e. a histogram is computed for each input image line separately.

The histogram itself consists of 2^(InputBitWidth) elements. Each element is the number of pixels for each tonal value. The first pixel at the histogram output is the number of pixels with tonal value zero.

Note that the input is blocked while the operator outputs its results. Therefore, small input images and or output for each image line reduce the bandwidth significantly.

**Image Restrictions**

- Empty Images

  Empty images, i.e., images with no pixels, are not allowed.

## 18.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, Image Input |
| **Output Link** | O, Data Output |

## 18.8.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 16] | auto❶ |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | auto❷ |
| **Max. Img Height** | any | auto❸ |

❶ The output bit width depends on the max image dimension link properties and the setting of parameter *AutoSync*.

For AutoSync = EoF the output bit width is

$$OutputBitWidth = \lceil log_2(Max.ImgWidth \times Max.ImgHeight + 1) \rceil$$

For AutoSync = EoL the output bit width is

$$OutputBitWidth = \lceil log_2(Max.ImgWidth + 1) \rceil$$

The output bit width must not exceed 64 Bit.

❷ The output image width is determined by the input bit width. Each possible pixel value of the input image is represented by the pixels of the output image.

$$OutputImageWidth = 2^{InputBitWidth}$$

❸ For parameter AutoSync = EoF, the output image height is one. For parameter AutoSync = EoL, the output image height is equal to the input image height.

## 18.8.3. Parameters

| **AutoSync** | |
|---|---|
| **Type** | static parameter |
| **Default** | EoL |
| **Range** | {EoL, EoF} |
| This parameter selects whether to output a histogram every line (setting EoL) or every frame (setting EoF). Note the parameter's influence on the output bit width. The output bit width must not exceed 64 Bit. | |

## 18.8.4. Examples of Use

The use of operator Histogram is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

# 18.9. Operator ModuloCount

**Operator Library: Accumulator**

The ModuloCounter counts the number of pixels, lines or frames at the input link. Its count sequence is restricted by the modulo divisor. The number of unique states that a counter may have before the sequence repeats itself is defined by the modulo divisor. The modulo divisor is defined with the *Divisor* parameter. For example, a modulo divisor of 4 will count "n MOD 4" i.e. 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3 ...

The image dimensions of the output frames are always equal to the image dimensions of the input frames. The pixel data at the output represent the counter result. Note that no pixel values are used. Only the pixel, line or frame index is considered for calculation.

The counter always starts from value zero. The *CountEntity* parameter can be configured to either count every pixel, line or frame of the input link. For example, if parameter *CountEntity* is set to *PIXEL*, the operator will increment its counter with every pixel of the input.

If *CountEntity* is set to *LINE*, the counter increments at the beginning of every line and hence, counts the lines of the input link. For example a modulo divisor of four will have the result that all pixels of line zero have value 0. The pixels of line one will have value one, etc. However, the pixels of line 4 will have the pixel values zero again.

If *CountEntity* is set to *FRAME*, the counter increments at the beginning of every frame.

In the following examples the operator's usage is illustrated. First example shows two successive frames which have an image width and image height of eight pixels. All pixel values are illustrated as numbers. Frame index 0 represents the initial frame, i.e. the first frame after acquisition start.

- Configuration: *Divisor* = 6, *CountEntity* = *PIXEL*, *AutoClear* = *EoF*

Frame0:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 |
| 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 |
| 4 | 5 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 |
| 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 |
| 4 | 5 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 |
| 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 |

Frame1:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 |
| 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 |
| 4 | 5 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 |
| 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 |
| 4 | 5 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 | 0 | 1 |
| 2 | 3 | 4 | 5 | 0 | 1 | 2 | 3 |

As can be seen in this example, the counter is incremented with every pixel when parameter *CountEntity* is set to *PIXEL*. The result values are in the range from zero to five, as the *Divisor* is set to six. In the second frame the operator starts from zero again, as the counter is reset at the end of the frame.

- Configuration: *Divisor* = 6, *CountEntity* = *LINE*, *AutoClear* = *NONE*

Frame0:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Frame1:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

In this example, the operator counts lines instead of pixels. Thus, every pixel of the image represents the line number. Here, parameter *AutoClear* is set to *NONE*. Therefore, the operator continues counting lines in the second frame without a reset at the first image line.

- Configuration: *Divisor* = 6, *CountEntity* = *FRAME*, *AutoClear* = *NONE*

Frame0:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Frame1:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

  The last example shows the counting of frames. Here, the value of every pixel represents the frame number.

If the parameter divisor is changed during acquisition, the new value is applied with the start of the next frame. For 1D input, the new value is applied for the next line. If 0D data is at the input, a new divisor value is applied immediately.

**Image Restrictions**

- Empty Images

  Empty images, i.e., images with no pixels, are not allowed.

## 18.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, Image Input |
| **Output Link** | O, Data Output |

## 18.9.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | [1, 63] |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 18.9.3. Parameters

| Divisor | |
|---|---|
| **Type** | static/dynamic write parameter |
| **Default** | 1 |
| **Range** | [1..2^OutputBitWidth] |
| This parameter defines the divisor of the modul operation. The divisor value has to be less equal than 2^OutputBitwidth. | |

| CountEntity | |
|---|---|
| **Type** | static parameter |
| **Default** | PIXEL |
| **Range** | {PIXEL, LINE, FRAME} |
| This parameter defines whether the operator counter is enabled for every pixel, every line or it is enabled once for each frame only.<br><br>Option FRAME is only available for image protocol VALT_IMAGE2D. LINE is only available for image protocols VALD_IMAGE2D and VALT_LINE1D. For image protocol VALT_PIXEL0D the parameter will always count pixel. | |

| AutoClear | |
|---|---|
| **Type** | static parameter |
| **Default** | NONE |
| **Range** | {EoL, EoF, NONE} |
| The AutoClear option allows a reset of the counter to zero at either the end of a line, the end of a frame or no clear at all is performed.<br><br>If CountEntity is set to PIXEL, all three options of this parameter can be selected. If CountEntity is set to LINE options EoF and NONE are available. For CountEntity FRAME, an auto clear of the operator is not possible.<br><br>If the VALT_LINE1D image protocol is used, no clear at the end of a frame is possible. In VALT_PIXEL0D, no auto clear of the operator is available. | |

### 18.9.4. Examples of Use

The use of operator ModuloCount is shown in the following examples:

- Section 9.2, ' Multiple DMA Channel Designs '

  Remove 9 out of 10 images.

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.4.2.3, 'Color Plane Separation Option 3 - Sequential with Operator ImageBufferMultiRoI'

  Sequential DMA output of the color planes. The color separations is performed using operator ImageBufferMultiROI.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.6.6, 'Image Grayscale Scope'

Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.14.2, 'Grid Overlay Fading'

  Examples - A grid is overlayed to the input images. The grid pixel value is determined from the input pixel value.

# 18.10. Operator Register

**Operator Library: Accumulator**

Register is a data storage element. Input pixels are stored and forwarded to the output while the *Capture* input is set to 1. If Capture is set to 0, than the previously stored value will be held.

It is possible to reset the register to the initial value at the and of a line, the end of a frame or never. This behavior is controlled using parameter *AutoClear*. An initial value can be defined using parameter *Init*. This initial value is applied on start-up.

Register operators are commonly used to store intermediate results.



**Resource Consumption**

The operator's FGPA resource consumption strongly increases with the parallelism. We recommend to use low parallelism at this operator.

In many use cases of this operator, the parallelism can be reduced by selecting the last parallel component using operator *SelectFromParallel* prior to this operator.

## 18.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>Capture, binary input to control the register |
| **Output Link** | O, Data Output |

## 18.10.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Capture | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D,<br>VALT_SIGNAL} | as I | as I |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FG_NONE | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 18.10.3. Parameters

| AutoClear | |
|---|---|
| **Type** | static parameter |

| AutoClear | |
|---|---|
| **Default** | EoL |
| **Range** | {EoL, EoF, None} |
| This parameter selects whether to reset the register to the initial value at end of line (EoL), end of frame (EoF) or never (None).<br><br>If image protocol VALT_LINE1D is used, the auto clear at the end of a frame is not possible. For VALT_PIXEL0D or VALT_SIGNAL, the parameter is disabled. | |

| Init | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 2^InputBitWidth-1] |
| This parameter defines the initialization value for the register operator at design start up, after a design reset, or after an auto clear event. The minimum value is 0. The maximum value can be derived from the pixel bit width at the input. Currently, no signed values can be edited. To set the parameter to a signed value enter the unsigned two's complements representation. For color and other multi component formats, the parameter Init defines the initialization value for all components, i.e. all components will be initialized with the same value. | |

## 18.10.4. Examples of Use

The use of operator Register is shown in the following examples:

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

# 18.11. Operator RowMax

**Operator Library: Accumulator**

This operator returns the maximum pixel value found inside each image row. Each input pixel value is replaced by the maximum pixel value found in the very same row so far. If the current pixel is a new maximum the output Link *IsMaxO* is 1, otherwise it is 0.



**Resource Consumption**

The operator's FGPA resource consumption strongly increases with the parallelism. We recommend to use low parallelism at this operator.

In many use cases of this operator, the parallelism can be reduced selecting the max parallel component prior to this operator. Use *SplitParallel*, *MergeKernel* and *MAX* for this operation.

Every new row restarts the maximum detection. Additionally, it is possible to clear the current row maximum manually when the *ClrI* link is set to 1. When a manual clear is not required, you must ensure a constant 0 at *ClrI*.

Often, this operator is used in conjunction with operator *RemovePixel* to use only the last column containing the results of the full frame. Note that the operator will require additional resources with increased parallelism. If only the maximum of each row without intermediate results is required, it is possible to determine the maximum of the parallel words first.

The following example shows the relation of input values to output values.

Frame I

| | | | |
|---|---|---|---|
| 0 | 2 | 3 | 8 |
| 3 | 4 | 5 | 5 |
| 5 | 3 | 8 | 5 |
| 1 | 1 | 1 | 1 |

Frame ClrI

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Frame O

| | | | |
|---|---|---|---|
| 0 | 2 | 3 | 8 |
| 3 | 4 | 5 | 5 |
| 5 | 5 | 8 | 8 |
| 1 | 1 | 1 | 1 |

**Image Restrictions**

- Image Size

  Varying line lengths and empty lines are allowed in non-empty images.

- Empty Images

  Empty images, i.e., images with no pixels, are allowed. The operator will output an empty image.

## 18.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Links** | O, Data Output<br>IsMaxO, Binary output to indicate a new maximum |

### 18.11.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O | Output Link IsMaxO |
|---|---|---|
| **Bit Width** | as I | 1 |
| **Arithmetic** | as I | unsigned |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | as I |
| **Color Flavor** | as I | as I |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

### 18.11.3. Parameters

None

### 18.11.4. Examples of Use

The use of operator RowMax is shown in the following examples:

- Section 13.1, 'Functional Example for Specific Operators of Library Accumulator and Library Logic' Examples - Demonstration of how to use the operator

# 18.12. Operator RowMin

**Operator Library: Accumulator**

This operator returns the minimum pixel value found inside each image row. The operator replaces each pixel value by the minimum value found in the very same row so far. If a current pixel is a new minimum, the output link *IsMinO* is 1, otherwise it is 0.



**Resource Consumption**

The operator's FGPA resource consumption strongly increases with the parallelism. We recommend to use low parallelism at this operator.

In many use cases of this operator, the parallelism can be reduced selecting the min parallel component prior to this operator. Use *SplitParallel*, *MergeKernel* and *MIN* for this operation.

With every new row the minimum detection starts again. Additionally it is possible to clear the current row minimum manually when the *ClrI* link is set to 1. When a manual clear is not required, you must ensure a constant 0 at *ClrI*.

Often, this operator is used in conjunction with operator *RemovePixel* to use only the last column containing the results of the full frame. Note that the operator will require additional resources with increased parallelism. If only the minimum of each row without intermediate results is required, it is possible to determine the minimum of the parallel words first.

The following example shows the relation of input values to output values.

Frame I

| | | | |
|---|---|---|---|
| 0 | 2 | 3 | 8 |
| 3 | 4 | 5 | 5 |
| 5 | 5 | 2 | 5 |
| 1 | 1 | 1 | 1 |

Frame ClrI

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Frame O

| | | | |
|---|---|---|---|
| 0 | 2 | 3 | 8 |
| 3 | 3 | 3 | 5 |
| 5 | 5 | 2 | 2 |
| 1 | 1 | 1 | 1 |

**Image Restrictions**

- Image Size

  Varying line lengths and empty lines are allowed in non-empty images.

- Empty Images

  Empty images, i.e., images with no pixels, are allowed. The operator will output an empty image.

## 18.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Links** | O, Data Output<br>IsMinO, Binary output to indicate a new maximum |

### 18.12.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O | Output Link IsMinO |
|---|---|---|
| **Bit Width** | as I | 1 |
| **Arithmetic** | as I | unsigned |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | as I |
| **Color Flavor** | as I | as I |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

### 18.12.3. Parameters

None

### 18.12.4. Examples of Use

The use of operator RowMin is shown in the following examples:

- Section 12.13.2, 'Print Inspection Example- Position Correction and Defect Detection Using Blob Based Template Matching'

  Examples- Geometric Transformation and Defect Detection

- Section 13.1, 'Functional Example for Specific Operators of Library Accumulator and Library Logic'

  Examples - Demonstration of how to use the operator

# 18.13. Operator RowSum

**Operator Library: Accumulator**

This operator calculates the sum of the pixel values for each image row. Each pixel value at the output represents the current sum of the image row. Thus, each pixel is replaced by the sum determined so far.



**Resource Consumption**

The operator's FGPA resource consumption strongly increases with the parallelism. We recommend to use low parallelism at this operator.

In many use cases of this operator, the parallelism can be reduced by adding up the parallel components prior this operator using *SplitParallel* and *ADD*.

With every new input row the sum for summations starts from zero. Additionally, it is possible to clear the current row sum manually when the *ClrI* link is set to 1. When a manual clearing is not required, you must ensure a constant 0 at *ClrI*.

The following example shows the relation of input values to output values.

Frame I

| | | | |
|---|---|---|---|
| 0 | 2 | 3 | 8 |
| 3 | 4 | 5 | 5 |
| 5 | 5 | 5 | 5 |
| 1 | 1 | 1 | 1 |

Frame ClrI

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

Frame O

| | | | |
|---|---|---|---|
| 0 | 2 | 3 | 8 |
| 3 | 7 | 12 | 5 |
| 5 | 10 | 15 | 20 |
| 1 | 1 | 1 | 1 |

Often, this operator is used in conjunction with operator *RemovePixel* to use only the last column containing the results of the full frame. Note that the operator will require additional resources with increased parallelism. If only the sum of each row without intermediate results is required, it is possible to determine the sum of the parallel words first.

**Image Restrictions**

- Image Size

  Varying line lengths and empty lines are allowed in non-empty images.

- Empty Images

  Empty images, i.e. images with no pixels, are allowed. The operator will output an empty image.

## 18.13.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, Image Input<br>ClrI, Clear Input |
| **Output Link** | O, Data Output |

## 18.13.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ClrI | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | 1 | auto❶ |

| Link Parameter | Input Link I | Input Link ClrI | Output Link O |
|---|---|---|---|
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | 1 | as I | as I |
| **Kernel Rows** | 1 | as I | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I | as I |
| **Color Format** | VAF_GRAY | as I | as I |
| **Color Flavor** | FL_NONE | as I | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The output bit width is automatically determined from the input bit width and maximum image width. The output bit width is determined by

$$OutputBitWidth = InputBitWidth \times \lceil log_2(Max.ImgWidth + 1) \rceil$$

The output bit width must not exceed 64 Bit.

## 18.13.3. Parameters

None

## 18.13.4. Examples of Use

The use of operator RowSum is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.10.3, 'Image Composition Using Exposure Fusion'

  Examples - ExposureFusion

# 19. Library Arithmetics




The *Arithmetics* library includes arithmetic operators such as ADD, SUB, COS, ShiftLeft, etc.

The following list summarizes all Operators of Library Arithmetics

| Operator Name | | Short Description | available since |
|---|---|---|---|
| ABS | ABS | Calculates the absolute value of the input | Version 1.1 |
| + | ADD | Calculates the sum of multiple inputs. | Version 1.1 |
| $\cos^{-1}$ | ARCCOS | Calculates the arc cosine of the input. | Version 1.1 |
| $\cot^{-1}$ | ARCCOT | Calculates the arccot of the input. | Version 1.1 |
| $\sin^{-1}$ | ARCSIN | Calculates the arc sine of the input. | Version 1.1 |
| $\tan^{-1}$ | ARCTAN | Calculates the arctan of the input. | Version 1.1 |
|  | ClipHigh | Limits the values to a parametrizable maximum. | Version 1.1 |
|  | ClipLow | Limits the values to a parametrizable minimum. | Version 1.1 |
| cos | COS | Calculates the cosine of the input. | Version 1.1 |
| cot | COT | Calculates the cotangent of the input. | Version 1.1 |
| ÷ | DIV | Divides the values at input link I1 by the values of input link I2 | Version 1.1 |
| * | MULT | Multiplies the values at input link I1 by the values of input link I2 | Version 1.1 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
| ≈ | RND | Performs a right shift with rounding. | Version 1.1 |
| n*x | SCALE | Performs a multiplication of the input with a parameterizable value. | Version 1.1 |
|  | ShiftLeft | Performs an arithmetic shift of the input data to the left. | Version 1.1 |
|  | ShiftRight | Performs an arithmetic shift of the input data to the right. | Version 1.1 |
| sin | SIN | Calculates the sine of the input. | Version 1.1 |
| √ | SQRT | Calculates the square root of the input. | Version 1.1 |
| - | SUB | Calculates the difference of the two input links. | Version 1.1 |
| tan | TAN | Calculates the tangent of the input. | Version 1.1 |

Table 19.1. Operators of Library Arithmetics

# 19.1. Operator ABS

**Operator Library: Arithmetics**

The operator calculates the absolute value of the input, i.e.

$$O = |I|$$

Note that the absolute value of the smallest possible negative value will require the same number of bits as its positive equivalent. Therefore, the output bit width is equal to the input bit width. For example, in 8 bit representation, the smallest value will be -128 while the maximum value is 127. After the absolute operation -128 becomes +128 what still requires 8 bit.

## 19.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 19.1.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [2, 64] | as I |
| **Arithmetic** | signed | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY, VAF_COLOR | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 19.1.3. Parameters

None

## 19.1.4. Examples of Use

The use of operator ABS is shown in the following examples:

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

- Section 12.8.1.2, 'Kirsch Filter'

  Examples - The Kirsch filter is a good edge detection filter for non directional edges.

- Section 12.8.1.3, 'Roberts Cross Gradient'

Examples - Roberts Cross Gradient filter example.

- Section 12.8.1.4, 'Sobel Gradient X'

  Examples - A Sobel filter in x-direction only.

- Section 12.8.1.5, 'Sobel Multi Gradient'

  Examples - A Sobel filter in all 4 directions.

- Section 12.8.4.2, 'Parallel Filters'

  Examples - An example of the use of two filters in parallel.

- Section 12.8.4.4, 'Filter for Line Scan Cameras'

  Examples - Explains how to implement a filter for line scan cameras.

- Section 12.8.5.2, 'Laplace Filter 3x3'

  Examples - A 3x3 Laplace filter.

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

# 19.2. Operator ADD

**Operator Library: Arithmetics**

The module ADD is calculating the sum over multiple input links. The number of input links has to be selected at the instantiation of the module.

## 19.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I[0], data input<br>I[n], n > 0, data input |
| **Output Link** | O, data output |

## 19.2.2. Supported Link Format

| Link Parameter | Input Link I[0] | Input Link I[n], n > 0 | Output Link O |
|---|---|---|---|
| **Bit Width** | {[1; 63] unsigned, [2; 63] signed} | {[1; 63] unsigned, [2; 63] signed} | auto❶ |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} | auto❷ |
| **Parallelism** | any | as I[0] | as I[0] |
| **Kernel Columns** | any | as I[0] | as I[0] |
| **Kernel Rows** | any | as I[0] | as I[0] |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I[0] | as I[0] |
| **Color Format** | VAF_GRAY | as I[0] | as I[0] |
| **Color Flavor** | FL_NONE | as I[0] | as I[0] |
| **Max. Img Width** | any | as I[0] | as I[0] |
| **Max. Img Height** | any | as I[0] | as I[0] |

❶ The output bit width is automatically determined from the maximum input link bit width. The output bit width is determined by

$$OutputBitWidth = \lceil log_2\left(No.ofInputs \times \left(2^{max.\ InputBitWidth} - 1\right)\right)\rceil$$

The output bit width must not exceed 64 Bit.

❷ If you use unsigned and signed values on the input links, you might need an extra bit on the output. The output arithmetic is unsigned if all of the inputs are unsigned. The output arithmetic is signed if at least one of the inputs uses a signed arithmetic.

## 19.2.3. Parameters

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |
| Parameter ImplementationType influences the implementation strategy of the operator, i.e., which logic elements are used for implementing the operator.<br><br>You can select one of the following values: | |

| ImplementationType |
| --- |
| **AUTO:** When the operator is instantiated, the optimal implementation strategy is selected automatically based on the parametrization of the connected links.<br><br>**EmbeddedALU:** The operator uses embedded arithmetic logic elements of the FPGA that are not LUT based.<br><br>**LUT:** The operator uses the LUT logic of the FPGA.<br><br> **Use AUTO in General**<br><br>Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT. |

### 19.2.4. Examples of Use

The use of operator ADD is shown in the following examples:

- Section 3.6.2, 'O-Type Networks'

  Synchronization Rules - The use of the operator in an O-type Network.

- Section 12.2.1, 'Adaptive Threshold'

  A binarization example for local adaptive thresholding. A kernel size of 8 by 8 pixel is used.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.8.1.3, 'Roberts Cross Gradient'

  Examples - Roberts Cross Gradient filter example.

- Section 12.8.1.5, 'Sobel Multi Gradient'

  Examples - A Sobel filter in all 4 directions.

- Section 12.8.4.2, 'Parallel Filters'

  Examples - An example of the use of two filters in parallel.

- Section 12.8.5.1, 'High Boost Sharpening Filter'

  Examples - A high boost Laplace filter for sharpening

- Section 12.9.5, 'Moments in Image Processing'

  Example - Calculates image moments orientation and eccentricity

- Section 12.14.2, 'Grid Overlay Fading'

  Examples - A grid is overlayed to the input images. The grid pixel value is determined from the input pixel value.

## 19.3. Operator ARCCOS

**Operator Library: Arithmetics**

**Implemented Function ist not arccos(x)**

The implemented function is -1 * arccos(x). See description.

The operator ARCCOS is calculating the arc cosine of the input. Note that not arccos(x) is implemented. Instead, the result is inverted i.e. -1 * arccos(x) is implemented.

The input range of arccos is [-1,1]. The operator maps the input interval range [-1, 1] to

$$\left[-2^{w_i-2}, 2^{w_i-2}\right]$$

where

$$w_i$$

is the bit width at the input. As can be seen, not the full value range of the input is used. If an input value is outside the allowed value range, the operator will clip the value to -1 or 1. Thus, the argument x of the arccos is determined by

$$x = \frac{InputValue}{2^{w_i-2}}$$

The operator calculates the inverse of the arccos i.e. the output value range is not [0, - π] it is [- π, 0]. The output value range of the operator in VisualApplets is mapped to

$$\left[-2^{w_o-1}, 0\right]$$

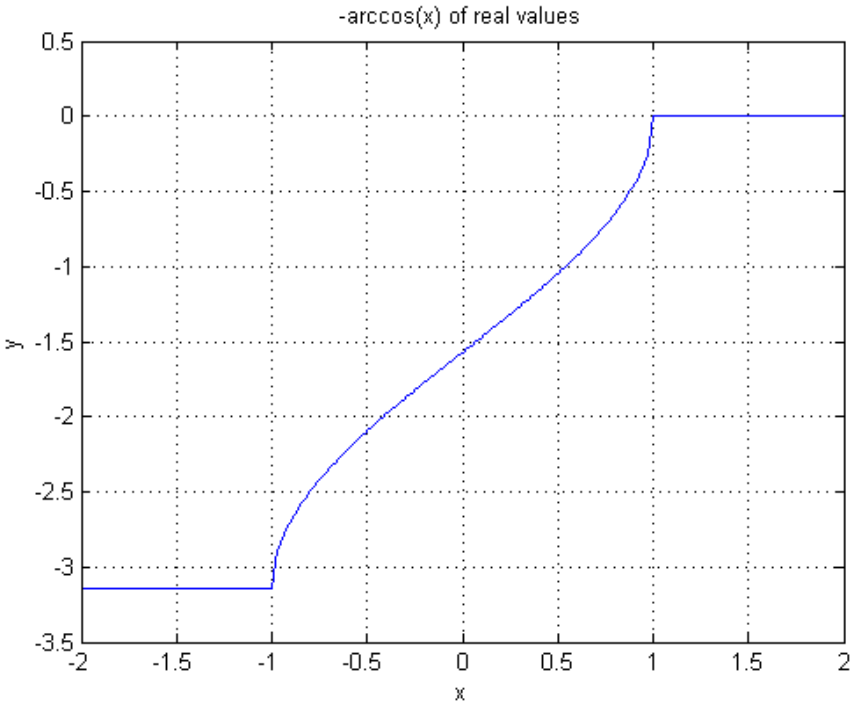
where

$$w_o$$

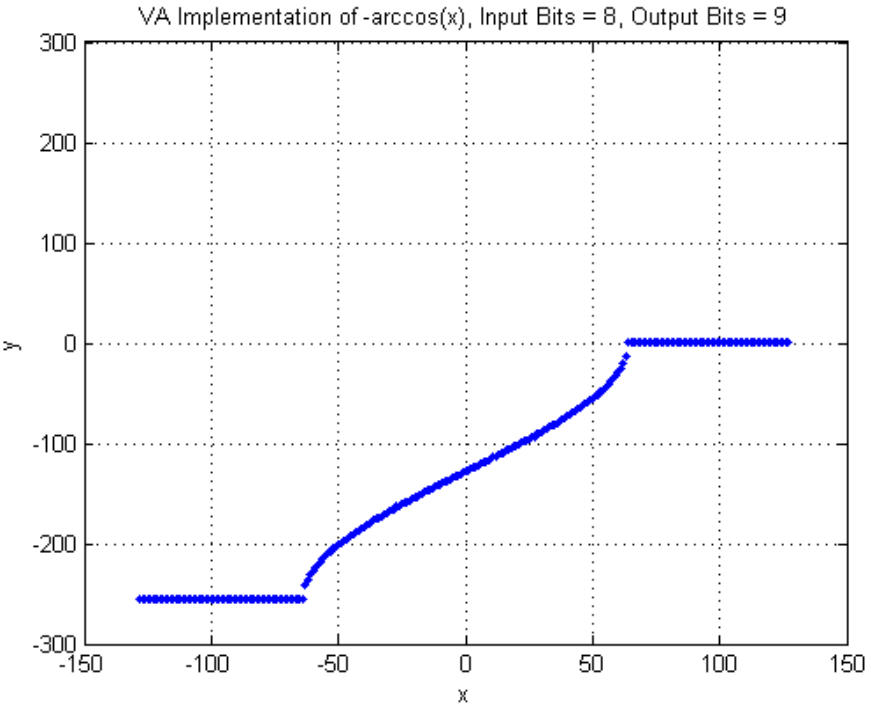
is the bit width at the output link. Thus the output value is

$$OutputValue = -\arccos(x) \times \frac{2^{w_o-1}}{\pi}$$

The following image shows the plot of the normal inverse arccos function.




In the next figure, the VisualApplets operator implementation is shown. Note the input and output bit widths.




### 19.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 19.3.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [8, 12] | [8,32] |
| **Arithmetic** | signed | signed |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 19.3.3. Parameters

None

### 19.3.4. Examples of Use

The use of operator ARCCOS is shown in the following examples:

- Section 13.7, 'Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions'

  Examples - Demonstration of how to use the operator

# 19.4. Operator ARCCOT

**Operator Library: Arithmetics**

The operator calculates the arccot of the input.

The input range of arccot is [∞, ∞]. This operator includes a parameter *Resolution* to define the fixed point resolution bits of the input values. The input range is given by the number of input bits =

$$\left[\frac{-2^{w_i-1}}{2^R}, \frac{2^{w_i-1}}{2^R}\right[$$

where

$$w_i$$

is the bit width at the input and R the resolution bits.

The argument x of the arccot function is determined by

$$x = \frac{InputValue}{2^R}$$

The results of the arccot function are in the range ]0, π[. The output value range of the operator in VisualApplets is mapped to

$$[0, 2^{b_o-1}]$$

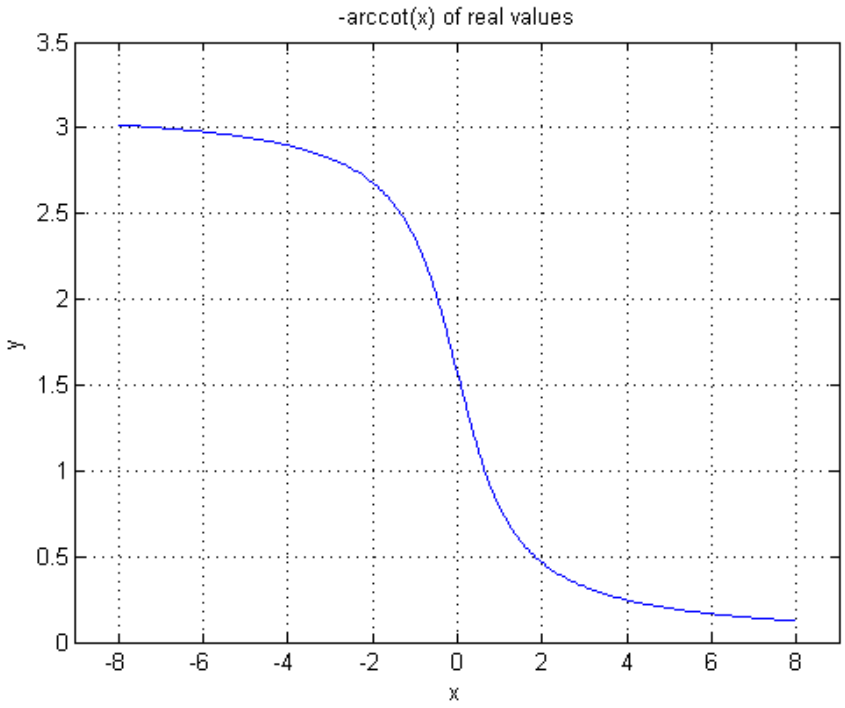
where

$$b_o$$

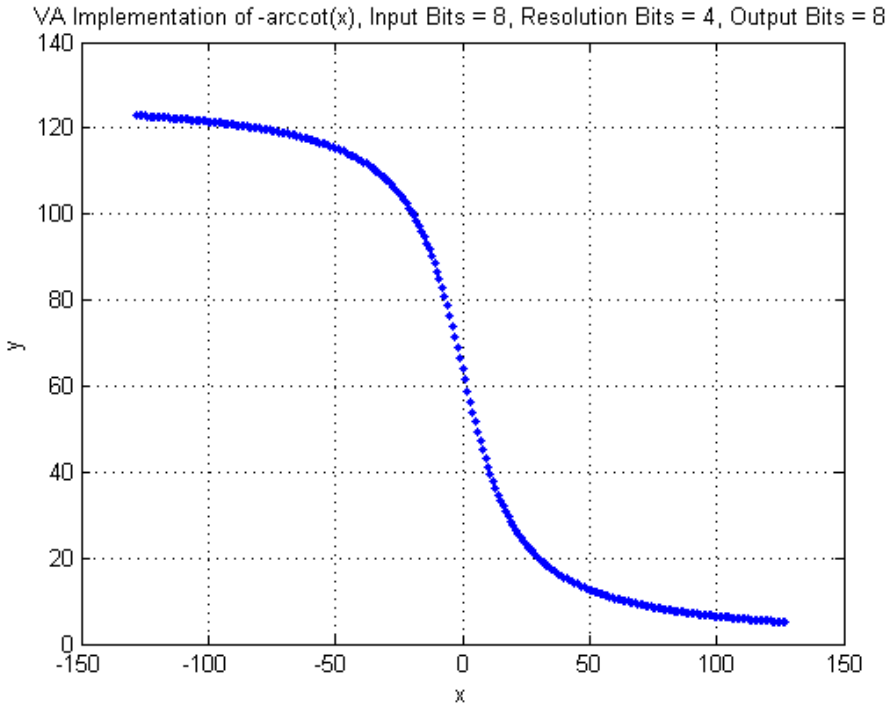
is the bit width at the output link. Thus the output value is

$$OutputValue = \operatorname{arccot}(x) \times \frac{2^{b_o-1}}{\pi}$$

The following image shows the plot of the arccot function.




In the next figure, the VisualApplets operator implementation is shown. Note the input and output bit widths.




Le't have a look at an input pixel value. For example -100. With the given resolution of four, the real value representation of the pixel value is -100/16. The arccot result of -100/16 is arccot(-100/16) = 2.98. In pixel value representation this result becomes 122 which is the same as shown in the plot.

### 19.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 19.4.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [8, 12] | [8,32] |
| **Arithmetic** | signed | signed |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 19.4.3. Parameters

| ResolutionBits | |
|---|---|
| **Type** | static parameter |
| **Default** | 8 |
| **Range** | [0..OutputBitWidth] |
| This parameter defines the accuracy of the input values as defined in the description above. | |

### 19.4.4. Examples of Use

The use of operator ARCCOT is shown in the following examples:

- Section 13.7, 'Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions'

  Examples - Demonstration of how to use the operator

# 19.5. Operator ARCSIN

**Operator Library: Arithmetics**

The operator ARCSIN is calculating the arc sine of the input.

The input range of arc sine is [-1,1]. The operator maps the input interval range [-1, 1] to $[-2^{w_i-2}, 2^{w_i-2}]$ where $w_i$ is the bit width at the input. As can be seen, not the full value range of the input is used. If an input value is outside the allowed value range, the operator will clip the value to -1 or 1. Thus, the argument x of the arc sine is determined by

$$x = \frac{InputValue}{2^{w_i-2}}$$

The operator's output range is [-π/2, π/2]. The output value range of the operator in VisualApplets is mapped to

$$[-2^{w_o-1}, 2^{w_o-1}]$$

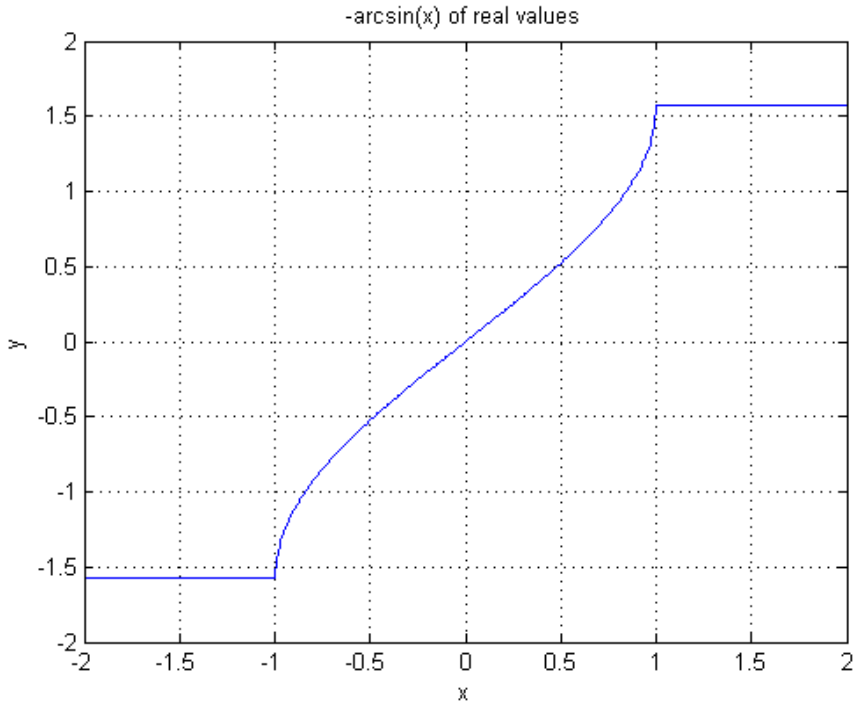
where

$$w_o$$

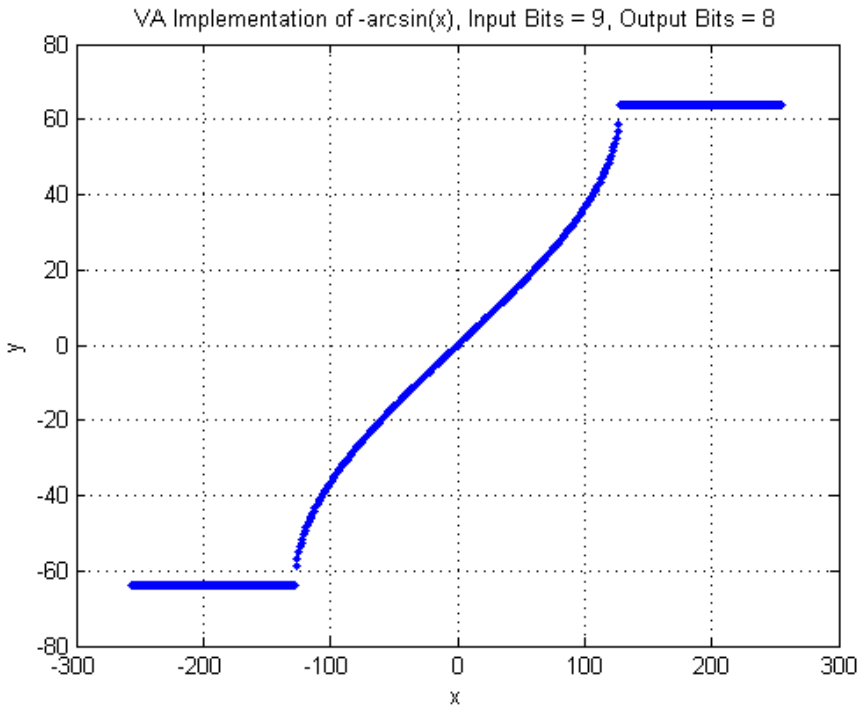
is the bit width at the output link. Thus the output value is

$$OutputValue = \arcsin(x) \times \frac{2^{w_o-1}}{\pi}$$

The following image shows the plot of the normal inverse arc sine function.




In the next figure, the VisualApplets operator implementation is shown. Note the input and output bit widths.




### 19.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 19.5.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [8, 12] | [8,32] |
| **Arithmetic** | signed | signed |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 19.5.3. Parameters

None

### 19.5.4. Examples of Use

The use of operator ARCSIN is shown in the following examples:

- Section 13.7, 'Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions'

  Examples - Demonstration of how to use the operator

# 19.6. Operator ARCTAN

**Operator Library: Arithmetics**

The operator calculates the arctan of the input.

The input range of arctan is [∞, ∞]. This operator includes a parameter *Resolution* to define the fixed point resolution bits of the input values. The input range is given by the number of input bits =

$$\left[\frac{-2^{w_i-1}}{2^R}, \frac{2^{w_i-1}}{2^R}\right[$$

where

$$w_i$$

is the bit width at the input and R the resolution bits.

The argument x of the arctan function is determined by

$$x = \frac{InputValue}{2^R}$$

The results of the arctan function are in the range ]- π/2, π/2[. The output value range of the operator in VisualApplets is mapped to

$$\left[-2^{b_o-2}, 2^{b_o-2}\right]$$

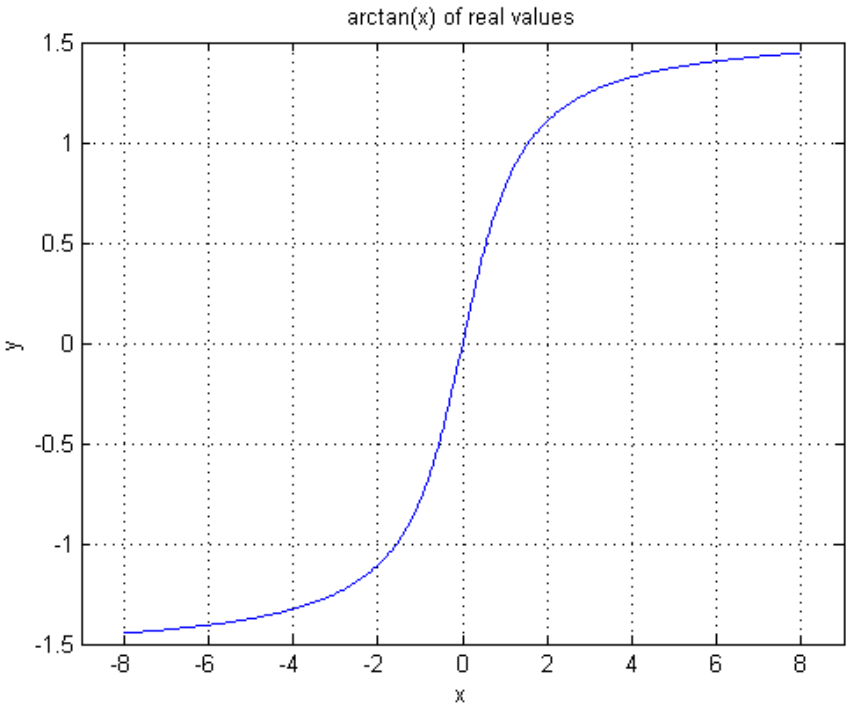
where

$$b_o$$

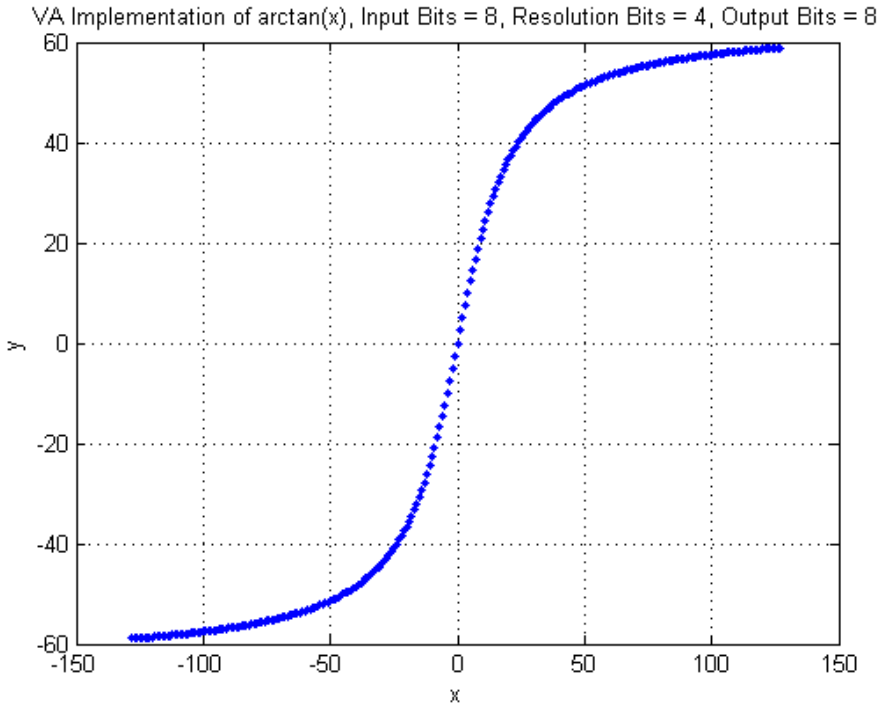
is the bit width at the output link. Thus the output value is

$$OutputValue = \arctan(x) \times \frac{2^{b_o-1}}{\pi}$$

The following image shows the plot of the arctan function.




In the next figure, the VisualApplets operator implementation is shown. Note the input and output bit widths.




Le't have a look at an input pixel value. For example -50. With the given resolution of four, the real value representation of the pixel value is -50/16. The arctan result of -50/16 is arctan(-50/16) = -1.26. In pixel value representation this result becomes -51,38 which is the same as shown in the plot.

### 19.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 19.6.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [8, 12] | [8,32] |
| **Arithmetic** | signed | signed |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 19.6.3. Parameters

| ResolutionBits | |
|---|---|
| **Type** | static parameter |
| **Default** | 8 |
| **Range** | [0, OutputBitWidth] |
| This parameter defines the accuracy of the input values as defined in the description above. | |

### 19.6.4. Examples of Use

The use of operator ARCTAN is shown in the following examples:

- Section 12.9.5, 'Moments in Image Processing'

  Example - Calculates image moments orientation and eccentricity

- Section 12.13.1, 'Histogram of Oriented Gradients (HOG)'

  Examples- Histogram of oriented Gradients

- Section 13.7, 'Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions'

  Examples - Demonstration of how to use the operator

# 19.7. Operator ClipHigh

**Operator Library: Arithmetics**

The module ClipHigh limits the values to a parametrizable maximum. If the input exceeds the maximum it is clipped, i.e.

$$O = \begin{cases} I & \text{if I} < \text{to} \\ to & \text{else} \end{cases}$$

## 19.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 19.7.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 63] signed | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 19.7.3. Parameters

| **to** | |
|---|---|
| **Type** | static parameter |
| **Default** | 127 |
| **Range** | range of the input link |
| This parameter defines the maximum value for the output link. | |

## 19.7.4. Examples of Use

The use of operator ClipHigh is shown in the following examples:

- Section 12.4.4, 'RGB White Balancing'

  Examples - The applet shows an example for white balancing on RGB images.

- Section 12.8.1.2, 'Kirsch Filter'

  Examples - The Kirsch filter is a good edge detection filter for non directional edges.

- Section 12.8.1.4, 'Sobel Gradient X'

  Examples - A Sobel filter in x-direction only.

- Section 12.8.1.5, 'Sobel Multi Gradient'

  Examples - A Sobel filter in all 4 directions.

- Section 12.8.3.2, 'Gaussian Filter 5x5'

  Examples - A Gauss filter using a 5x5 kernel.

- Section 12.8.5.2, 'Laplace Filter 3x3'

  Examples - A 3x3 Laplace filter.

- Section 12.10.3, 'Image Composition Using Exposure Fusion'

  Examples - ExposureFusion

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

# 19.8. Operator ClipLow

**Operator Library: Arithmetics**

The module ClipLow limits the values to a parametrizable minimum. If the input falls below the minimum it is clipped, i.e.

$$O = \begin{cases} I & \text{if I} > \text{from} \\ from & \text{else} \end{cases}$$

## 19.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 19.8.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 63] signed | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 19.8.3. Parameters

| **from** | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | range of the input link |
| This parameter defines the minimum value for the output link. | |

## 19.8.4. Examples of Use

The use of operator ClipLow is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

## 19.9. Operator COS

**Operator Library: Arithmetics**

The operator calculates the cosine of the input.

The input range of the cosine function is [∞, ∞]. Because of the periodicity of the cosine function the input range of the VisualApplets operator is limited to [п, п[ i.e. the minimum value at the input is -п and the maximum value at the input plus 1 is п Thus, it it not possible to have the value +п at the input.

The argument x of the cosine function is therefore determined by

$$x = InputValue \times \frac{\pi}{2^{w_i - 1}}$$

where

$$w_i$$

is the bit width at the input link.

The results of the cosine function are in the range [-1, 1]. The output value range of the operator in VisualApplets is mapped to

$$\left[-2^{w_o-2}, 2^{w_o-2}\right]$$

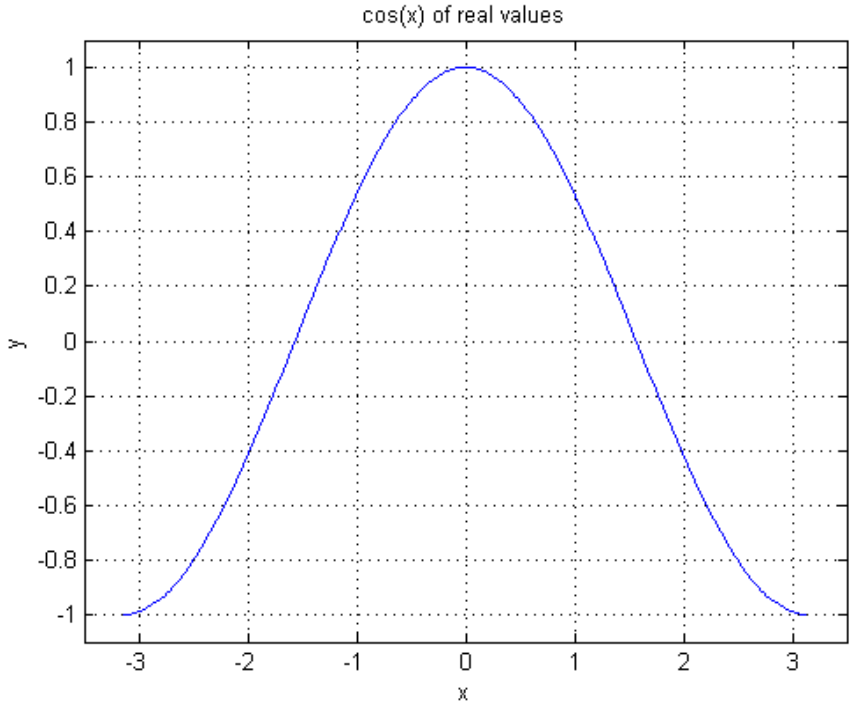
where

$$w_o$$

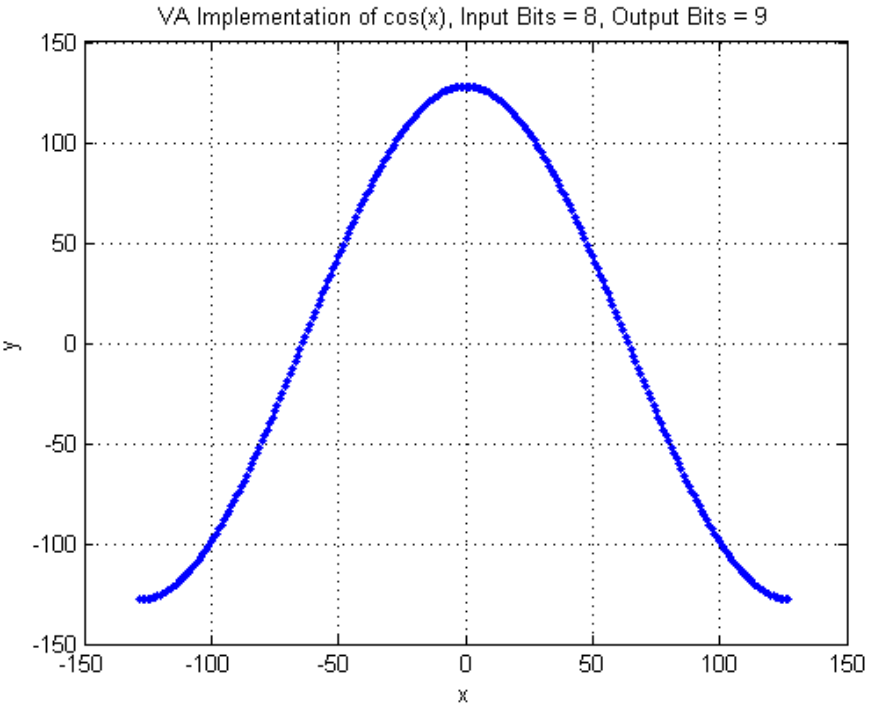
is the bit width at the output link. Thus the output value is

$$OutputValue = \cos(x) \times 2^{w_o-2}$$

The following image shows the plot of the cosine function.




In the next figure, the VisualApplets operator implementation is shown. Note the input and output bit widths.




Le't have a look at an input pixel value. For example -50. With the given input bit width of 8, the real value representation of the pixel value is -50 * pi / 128. The cosine result will then be 0.33. In pixel value representation this result becomes 43 which is the same as shown in the plot.

### 19.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 19.9.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [8, 12] | [8, 32] |
| **Arithmetic** | signed | signed |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 19.9.3. Parameters

None

### 19.9.4. Examples of Use

The use of operator COS is shown in the following examples:

- Section 12.9.3.2.4, 'Geometric Transformation and Distortion Correction'

  Examples- Geometric Transformation and Distortion Correction using **PixelReplicator**

- Section 13.7, 'Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions'

  Examples - Demonstration of how to use the operator

# 19.10. Operator COT

**Operator Library: Arithmetics**

The operator COT calculates the cotangent of the input.

The input range of the cotangent function is [-∞, ∞]. Because of the periodicity of the cotangent function the input range of the VisualApplets operator is limited to [-π, π[ i.e. the minimum value at the input is -π and the maximum value at the input plus 1 is π. Thus, it it not possible to have the value +π at the input.

The argument x of the cotangent function is therefore determined by

$$x = InputValue \times \frac{\pi}{2^{w_i - 1}}$$

where

$$w_i$$

is the bit width at the input link.

The results of the cotangent function are in the range [-∞, ∞]. This operator inlcudes a parameter *ResolutionBits* to define the fixed point resolution bits of the output values. The output range is therefore given by the number of output bits and the resolution bits =

$$\left[\frac{-2^{w_o-2}}{2^R}, \frac{2^{w_o-2}-1}{2^R}\right[$$

where

$$w_o$$

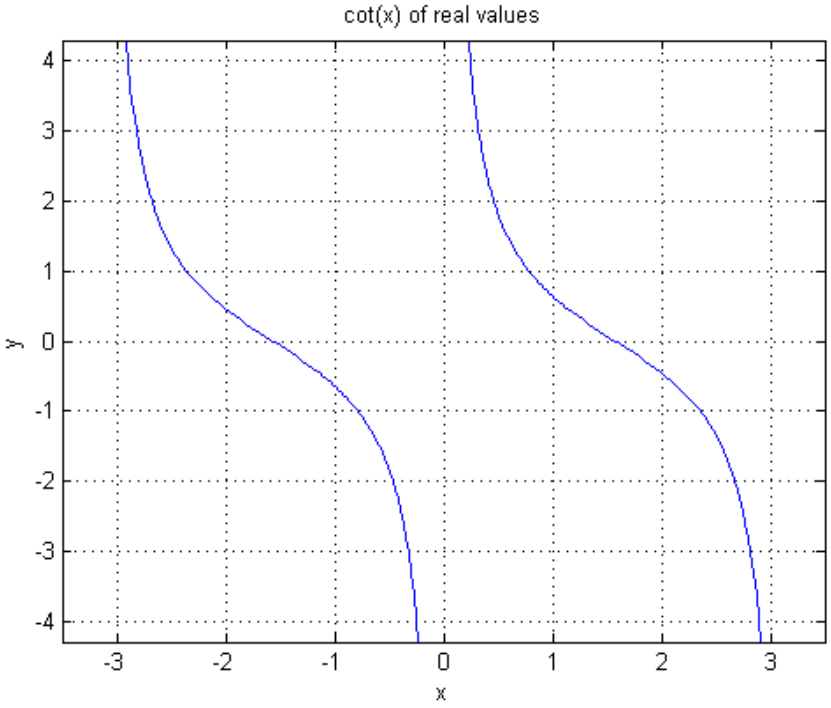
is the bit width at the output and R the resolution bits. Thus, the output value is

$$OutputValue = \cot(x) \times 2^R$$

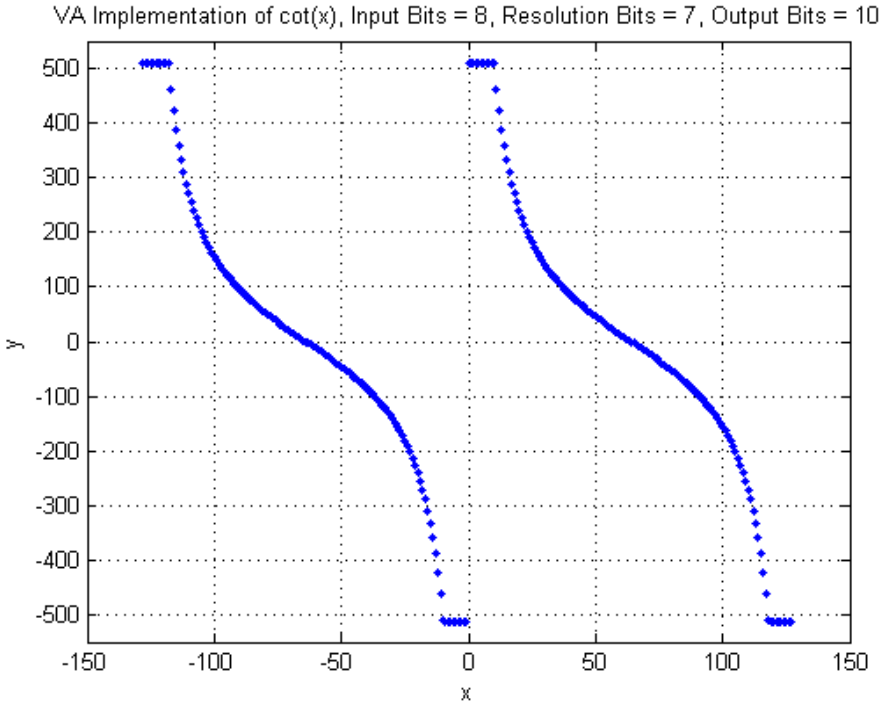
The values -π/2 and π/2 are not defined for the tangent function. The VisualApplets operator will clip these values to the maximum or minimum possible values at the output. Moreover, the result of the cotangent function can be out of the output value range. In this case, the results will be clipped to the maximum or minimum possible values, too. Thus,

$$OutputValue = \begin{cases} 2^{w_o-1} - 1 & \text{if } x = -\pi/2 \text{ or } tan(x) \times 2^R > 2^{w_o-1} - 1 \\ -2^{w_o-1} & \text{if } x = \pi/2 \text{ or } tan(x) \times 2^R < -2^{w_o-1} \\ \cot(x) \times 2^R & \text{else} \end{cases}$$

The following image shows the plot of the normal cotangent function.




In the next figure, the VisualApplets operator implementation is shown. Note the input and output bit widths.




## 19.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |

| Property | Value |
|---|---|
| **Output Link** | O, data output |

## 19.10.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [8, 12] | [8,32] |
| **Arithmetic** | signed | signed |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 19.10.3. Parameters

None

## 19.10.4. Examples of Use

The use of operator COT is shown in the following examples:

- Section 13.7, 'Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions'

  Examples - Demonstration of how to use the operator

# 19.11. Operator DIV

**Operator Library: Arithmetics**

The operator DIV divides the values at input link I1 by the values at input link I2. At output link O, the integer result of the division is provided. At output link R the remainder of the division is provided.

A division by zero is undefined at both outputs.



**Resources**

A division requires many FPGA resources. Try to use a minimized parallelism when performing a division. For divisions by a power of two value (2^n) use the shift operator *ShiftRight* instead.

## 19.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I1, dividend / numerator input<br>I2, divisor / denominator input |
| **Output Links** | O, integer quotient output<br>R, remainder output |

## 19.11.2. Supported Link Format

| Link Parameter | Input Link I1 | Input Link I2 |
|---|---|---|
| **Bit Width** | [1, 64] | [1, 64]❶ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O | Output Link R |
|---|---|---|
| **Bit Width** | auto❷ | as I2 |
| **Arithmetic** | as I | as I |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | as I |
| **Color Flavor** | as I | as I |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

❶ The bit width of the input I2 (divisor) has to be less equal than the bitwidth of the input at link I1 (dividend).
❷ For unsigned inputs the bit width at output O is equal to the bitwidth at I1. For signed inputs, the bit width is equal to the bit width at I1 + 1.

### 19.11.3. Parameters

None

### 19.11.4. Examples of Use

The use of operator DIV is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.9.5, 'Moments in Image Processing'

  Example - Calculates image moments orientation and eccentricity

- Section 12.9.7, 'Shear of an Image'

  Example - Line Shear example with linear interpolation.

# 19.12. Operator MULT

**Operator Library: Arithmetics**

Operator MULT multiplies the values at input link I1 by the values at input link I2. At output link O, the result of the multiplication is provided.

Operator MULT supports asymmetric arithmetic types on its inputs, i.e., SIGNED and UNSIGNED may be mixed up so that you can multiply an unsigned multiplier with a signed multiplicand or a signed multiplier with an unsigned multiplicand as well as a signed multiplier with a signed multiplicand or an unsigned multiplier with an unsigned multiplicand.



**Resources**

A multiplication requires many FPGA resources. For scaling with a constant use operator *SCALE* or *ShiftLeft* instead. Moreover, most frame grabbers include embedded arithmetic logic units (embedded multipliers). Use operator *HWMULT* to save resources.

## 19.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I1, multiplicand 1<br>I2, multiplicand 2 |
| **Output Link** | O, multiplication result |

## 19.12.2. Supported Link Format

| Link Parameter | Input Link I1 | Input Link I2 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 32] unsigned, [2, 32] signed | [1, 32] unsigned, [2, 32] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} | {unsigned, signed}❷ |
| **Parallelism** | any | as I1 | as I1 |
| **Kernel Columns** | any | as I1 | as I1 |
| **Kernel Rows** | any | as I1 | as I1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I1 | as I1 |
| **Color Format** | VAF_GRAY | as I1 | as I1 |
| **Color Flavor** | FL_NONE | as I1 | as I1 |
| **Max. Img Width** | any | as I1 | as I1 |
| **Max. Img Height** | any | as I1 | as I1 |

❶ The output bit width is the sum of the input bit widths i.e. bit width at I1 + bit width at I2.
❷ If I1 or I2 is set to *signed*, O must be set to *signed* as well.

## 19.12.3. Parameters

None

## 19.12.4. Examples of Use

The use of operator MULT is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

# 19.13. Operator RND

**Operator Library: Arithmetics**

The operator RND performs a right shift with rounding and clipping to the output bit width. The number of bits to round is specified with parameter *bits2Round*.

Round half away from zero is the implemented rounding method. Color components are rounded separately.

**Examples**

- Input = 4 Bit unsigned, Input Value = 7, bits2Round = 2

  Output = 2 Bit unsigned, Output Value = 2

- Input = 5 Bit signed, Input Value = -13, bits2Round = 2

  Output = 3 Bit signed, Output Value = -3

- Input = 4 Bit unsigned, Input Value = 15, bits2Round = 2

  Output = 2 Bit unsigned, Output Value = 3 Note: The output has been clipped to the maximum possible value.

## 19.13.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 19.13.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The bit width at the output is the input bit width minus parameter value of *bits2Round*.

## 19.13.3. Parameters

| **bits2Round** | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |

| bits2Round | |
|---|---|
| **Range** | [0, BitWidth(I)-1] for unsigned<br><br>(BitWidth(I) - bits2Round > 1) for signed |
| This parameter defines by how many bits the input is shifted to the right. | |

## 19.13.4. Examples of Use

The use of operator RND is shown in the following examples:

- Section 12.2.1, 'Adaptive Threshold'

  A binarization example for local adaptive thresholding. A kernel size of 8 by 8 pixel is used.

- Section 12.4.1.4, 'Bayer 3x3 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Moreover, a white balancing for color correction is added.

# 19.14. Operator SCALE

**Operator Library: Arithmetics**

The operator performs a multiplication of the input with a parameterizable value. The multiplicand can be defined using parameter *ScaleFactor*. The range of *ScaleFactor* is defined by parameter *ScaleFactorMaxBits* if *ScaleFactor* is a dynamic parameter. Only positive values are allowed for the scale factor.



**Less Resources for Static Parameter**

The operator requires less FPGA resources if the parameter *ScaleFactor* is set to static. Moreover, users should consider using operator *HWMULT* together with *CONST* to save resources.

## 19.14.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 19.14.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 32] unsigned, [2, 32] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The bit width at the output is the input bit width + parameter *ScaleFactorMaxBits* if parameter *ScaleFactor* is set to dynamic. If the parameter is set to static, the output bit width is

$$OutputBitWidth = \log_2\left(ScaleFactor * 2^{InputBitWidth} - 1\right)$$

## 19.14.3. Parameters

| ScaleFactorMaxBits | |
|---|---|
| **Type** | static parameter |
| **Default** | 8 |
| **Range** | [1, 32] |
| Using this parameter, the range of parameter *ScaleFactor* is defined. If *ScaleFactor* is set to static, this parameter is disabled | |

| ScaleFactor | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 1 |
| **Range** | [0, 2^ScaleFactorMaxBits - 1] if dynamic |
| This parameter defines the factor for the multiplication with the operator's input. The range of this parameter is [0, 1^ScaleFactorMaxBits - 1], if parameter *ScaleFactor* is dynamic. Otherwise, the range of this parameter is [1, 2^32 - 1]. | |

## 19.14.4. Examples of Use

The use of operator SCALE is shown in the following examples:

- Section 9.2, ' Multiple DMA Channel Designs '

  Threshold binarization

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.2, 'Blob 2D'

  Examples - Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels.

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.8.1.2, 'Kirsch Filter'

  Examples - The Kirsch filter is a good edge detection filter for non directional edges.

- Section 12.8.1.3, 'Roberts Cross Gradient'

  Examples - Roberts Cross Gradient filter example.

- Section 12.8.1.4, 'Sobel Gradient X'

  Examples - A Sobel filter in x-direction only.

- Section 12.8.1.5, 'Sobel Multi Gradient'

  Examples - A Sobel filter in all 4 directions.

- Section 12.8.2.1, 'Close'

  Examples - Shows the implementation of a morphological close applied to binary images.

- Section 12.8.2.3, 'Open'

  Examples - Shows the implementation of a morphological open applied to binary images.

- Section 12.8.3.1, 'Averaging 3x3'

Examples - A simple 3x3 box filter.

- Section 12.8.3.2, 'Gaussian Filter 5x5'

  Examples - A Gauss filter using a 5x5 kernel.

- Section 12.8.4.3, 'Filter Sub Kernels'

  Examples - Shows how to extract a sub kernel from a filter to obtain the original image data. This example performs a simple local adaptive binarization.

- Section 12.8.5.1, 'High Boost Sharpening Filter'

  Examples - A high boost Laplace filter for sharpening

- Section 12.8.5.2, 'Laplace Filter 3x3'

  Examples - A 3x3 Laplace filter.

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

# 19.15. Operator ShiftLeft

**Operator Library: Arithmetics**

The operator performs an arithmetic shift of the input data to the left. The number of bits to be shifted is defined using parameter *Shift*. Each bit at the input is left shifted by the parameterized number of bits. The newly inserted least significant bits will have value zero.

Bit shifting is usually used to scale a value by a power of two value. For example a left shift by one bit will double the value. A left shift by 2 bits will quadruple the input value.

Each color component is shifted separately.

## 19.15.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 19.15.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The bit width at the output is the input bit width plus the value of parameter *Shift*.

## 19.15.3. Parameters

| **Shift** | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 64 - Input BitWidth] |
| This parameter defines the number of bits by which the input is left shifted. | |

## 19.15.4. Examples of Use

The use of operator ShiftLeft is shown in the following examples:

- Figure 9.8, 'ShiftLeft Operator Added for 16Bit Output'

  Tutorial - User *ShiftLeft* to change DMA bit width.

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.9.5, 'Moments in Image Processing'

  Example - Calculates image moments orientation and eccentricity

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

# 19.16. Operator ShiftRight

**Operator Library: Arithmetics**

The operator performs an arithmetic shift of the input data to the right. The number of bits to be shifted is defined using parameter *Shift*. Each bit at the input is right shifted by the parameterized number of bits.

Bit shifting is usually used to scale a value by a power of two value. For example a right shift by one bit will divide the input value by 2. A right shift by 2 bits will divide the input value by 4.

Each color component is shifted separately.

## 19.16.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 19.16.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The bit width at the output is the input bit width minus the value of parameter *Shift*.

## 19.16.3. Parameters

| **Shift** | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, Input BitWidth - 1] for unsigned<br>[0, Input BitWidth - 2] for signed |
| This parameter defines the number of bits by which the input is right shifted. | |

## 19.16.4. Examples of Use

The use of operator ShiftRight is shown in the following examples:

- Figure 9.8, 'ShiftLeft Operator Added for 16Bit Output'

Tutorial - User *ShiftLeft* to change DMA bit width.

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend'

  Tutorial - From equation to implementation. Explanation on how to implement the overlay blend.

- Section 12.2.2, 'Auto Threshold Mean'

  Determines the mean value of an image and used the value as threshold value for the next image processed.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

- Section 12.8.1.2, 'Kirsch Filter'

  Examples - The Kirsch filter is a good edge detection filter for non directional edges.

- Section 12.8.1.4, 'Sobel Gradient X'

  Examples - A Sobel filter in x-direction only.

- Section 12.8.1.5, 'Sobel Multi Gradient'

  Examples - A Sobel filter in all 4 directions.

- Section 12.8.3.1, 'Averaging 3x3'

  Examples - A simple 3x3 box filter.

- Section 12.8.3.2, 'Gaussian Filter 5x5'

  Examples - A Gauss filter using a 5x5 kernel.

- Section 12.8.4.1, 'Filter Basics'

  Examples - Explains the implementation of filters.

- Section 12.8.4.2, 'Parallel Filters'

  Examples - An example of the use of two filters in parallel.

- Section 12.8.4.3, 'Filter Sub Kernels'

  Examples - Shows how to extract a sub kernel from a filter to obtain the original image data. This example performs a simple local adaptive binarization.

- Section 12.8.4.4, 'Filter for Line Scan Cameras'

  Examples - Explains how to implement a filter for line scan cameras.

- Section 12.8.5.2, 'Laplace Filter 3x3'

  Examples - A 3x3 Laplace filter.

- Section 12.9.2, 'Downsampling 3x3'

Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.9.5, 'Moments in Image Processing'

  Example - Calculates image moments orientation and eccentricity

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

# 19.17. Operator SIN

**Operator Library: Arithmetics**

The operator calculates the sine of the input.

The input range of the sine function is [-∞, ∞]. Because of the periodicity of the sine function the input range of the VisualApplets operator is limited to [-π, π] i.e. the minimum value at the input is -π and the maximum value at the input plus 1 is π. Thus, it it not possible to have the value +π at the input.

The argument x of the sine function is therefore determined by

$$x = InputValue \times \frac{\pi}{2^{w_i - 1}}$$

where

$$w_i$$

is the bit width at the input link.

The results of the sine function are in the range [-1, 1]. The output value range of the operator in VisualApplets is mapped to

$$[-2^{w_o-2}, 2^{w_o-2}]$$

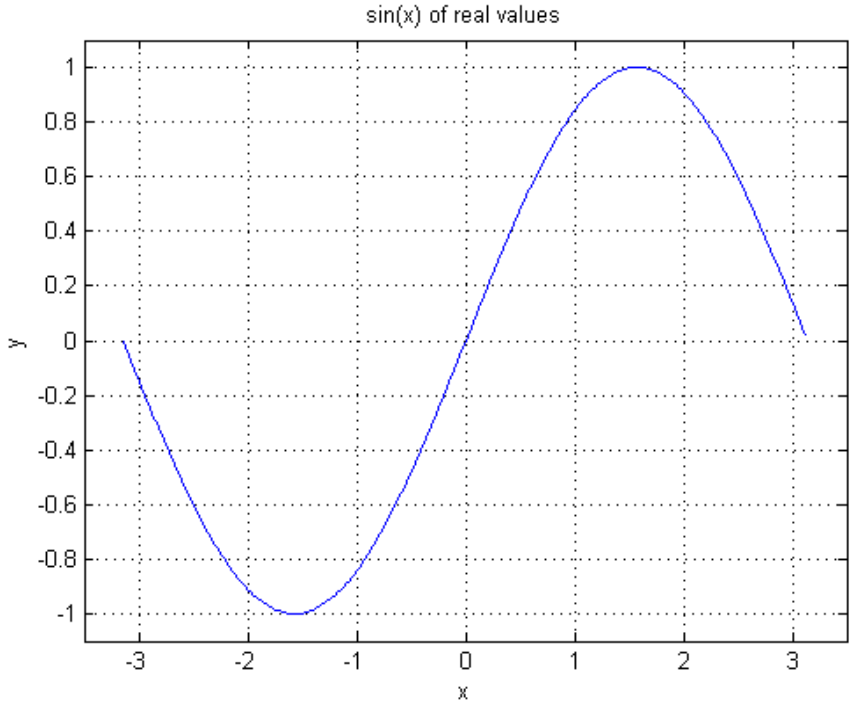
where

$$w_o$$

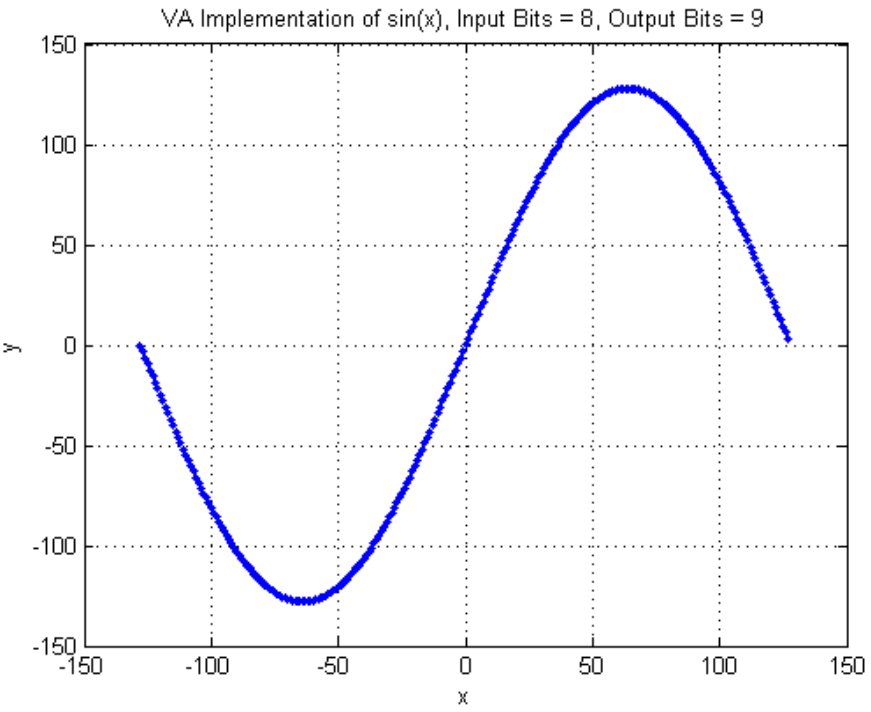
is the bit width at the output link. Thus the output value is

$$OutputValue = \sin(x) \times 2^{w_o-2}$$

The following image shows the plot of the sine function.




In the next figure, the VisualApplets operator implementation is shown. Note the input and output bit widths.




Le't have a look at an input pixel value. For example -50. With the given input bit width of 8, the real value representation of the pixel value is -50 * pi / 128. The sine result will then be -0.94. In pixel value representation this result becomes -121 which is the same as shown in the plot.

### 19.17.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 19.17.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [8, 12] | [8, 32] |
| **Arithmetic** | signed | signed |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 19.17.3. Parameters

None

### 19.17.4. Examples of Use

The use of operator SIN is shown in the following examples:

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

- Section 12.9.3.2.4, 'Geometric Transformation and Distortion Correction'

  Examples- Geometric Transformation and Distortion Correction using **PixelReplicator**

- Section 13.7, 'Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions'

  Examples - Demonstration of how to use the operator

# 19.18. Operator SQRT

**Operator Library: Arithmetics**

The operator calculates the square root of the input. The result is rounded to the next integer value.

## 19.18.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 19.18.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | {8, 12, 16, 20, 24, 28, 32} | auto❶ |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The bit width at the output is

$$OutputBitWidth = \frac{InputBitWidth}{2} + 1$$

## 19.18.3. Parameters

None

## 19.18.4. Examples of Use

The use of operator SQRT is shown in the following examples:

- Section 12.9.3.2.4, 'Geometric Transformation and Distortion Correction'

  Examples- Geometric Transformation and Distortion Correction using **PixelReplicator**
- Section 12.9.3.2.5, 'Distortion Correction'

  Examples- Distortion Correction

# 19.19. Operator SUB

**Operator Library: Arithmetics**

Operator SUB calculates the difference between input link I1 and input link I2 i.e. O = I1 - I2

## 19.19.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I1, data input<br>I2, data input |
| **Output Link** | O, data output |

## 19.19.2. Supported Link Format

| Link Parameter | Input Link I1 | Input Link I2 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 62] unsigned, [2, 62] signed | [1, 62] unsigned, [2, 62] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} | signed |
| **Parallelism** | any | as I1 | as I1 |
| **Kernel Columns** | any | as I1 | as I1 |
| **Kernel Rows** | any | as I1 | as I1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I1 |
| **Color Format** | VAF_GRAY | VAF_GRAY | as I1 |
| **Color Flavor** | FL_NONE | FL_NONE | as I1 |
| **Max. Img Width** | any | as I1 | as I1 |
| **Max. Img Height** | any | as I1 | as I1 |

❶ The output bit width is automatically determined from the input link bit widths. The output bit width is determined by

$$OutputBitWidth = \mathrm{Max}\{BitWidth(I1), BitWidth(I2)\} + 1$$

## 19.19.3. Parameters

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType influences the implementation strategy of the operator, i.e., which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy is selected automatically based on the parametrization of the connected links.

**EmbeddedALU:** The operator uses embedded arithmetic logic elements of the FPGA that are not LUT based.

| ImplementationType |
| --- |
| **LUT:** The operator uses the LUT logic of the FPGA. |



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

## 19.19.4. Examples of Use

The use of operator SUB is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

- Section 12.9.5, 'Moments in Image Processing'

  Example - Calculates image moments orientation and eccentricity

- Section 12.9.6, 'Line Mirror'

  Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

# 19.20. Operator TAN

**Operator Library: Arithmetics**

The operator TAN calculates the tangent of the input.

The input range of the tangent function is [-∞, ∞]. Because of the periodicity of the tangent function the input range of the VisualApplets operator is limited to [-π, π[ i.e. the minimum value at the input is -π and the maximum value at the input plus 1 is π. Thus, it it not possible to have the value +π at the input.

The argument x of the tangent function is therefore determined by

$$x = InputValue \times \frac{\pi}{2^{w_i - 1}}$$

where

$$w_i$$

is the bit width at the input link.

The results of the tangent function are in the range [-∞, ∞]. This operator includes a parameter *ResolutionBits* to define the fixed point resolution bits of the output values. The output range is therefore given by the number of output bits and the resolution bits =

$$\left[\frac{-2^{w_o-2}}{2^R}, \frac{2^{w_o-2}-1}{2^R}\right[$$

where

$$w_o$$

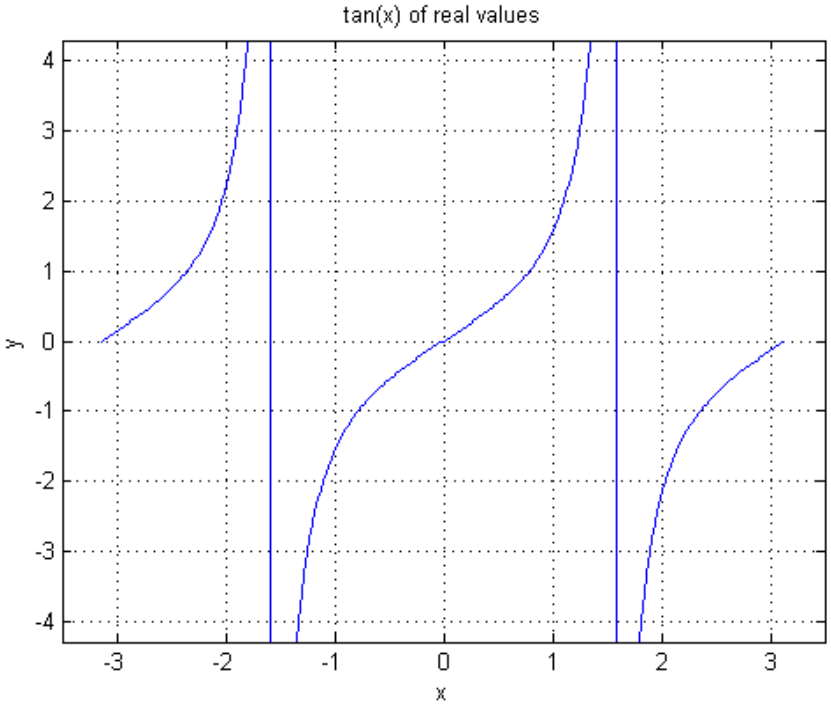
is the bit width at the output and R the resolution bits. Thus, the output value is

$$OutputValue = \tan(x) \times 2^R$$

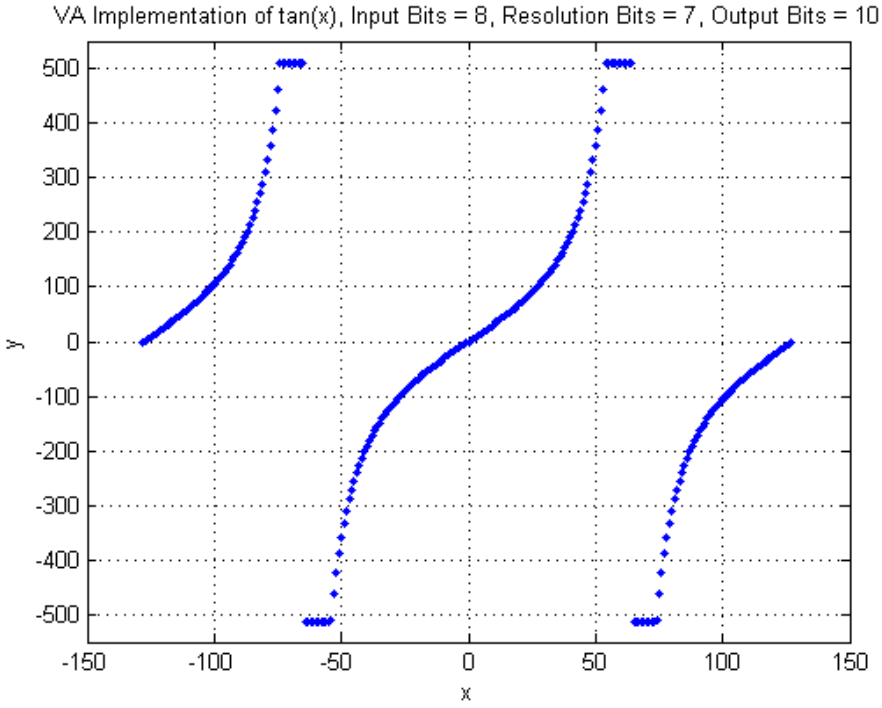
The values -π/2 and π/2 are not defined for the tangent function. The VisualApplets operator will clip these values to the maximum or minimum possible values at the output. Moreover, the result of the tangent function can be out of the output value range. In this case, the results will be clipped to the maximum or minimum possible values, too. Thus,

$$OutputValue = \begin{cases} 2^{w_o-1} - 1 & \text{if } x = -\pi/2 \text{ or } tan(x) \times 2^R > 2^{w_o-1} - 1 \\ -2^{w_o-1} & \text{if } x = \pi/2 \text{ or } tan(x) \times 2^R < -2^{w_o-1} \\ \tan(x) \times 2^R & \text{else} \end{cases}$$

The following image shows the plot of the normal tangent function.




In the next figure, the VisualApplets operator implementation is shown. Note the input and output bit widths.




## 19.20.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |

| Property | Value |
|---|---|
| **Output Link** | O, data output |

## 19.20.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [8, 12] | [8,32] |
| **Arithmetic** | signed | signed |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 19.20.3. Parameters

| ResolutionBits | |
|---|---|
| **Type** | static parameter |
| **Default** | 8 |
| **Range** | [0, OutputBitWidth] |
| This parameter defines the accuracy of the input values as defined in the description above. | |

## 19.20.4. Examples of Use

The use of operator TAN is shown in the following examples:

- Section 13.7, 'Functional Example for Specific Operators of Library Arithmentics: Trigonometric Functions'

  Examples - Demonstration of how to use the operator

# 20. Library Base




The *Base* library include common function operators required in most VisualApplets designs.

The following list summarizes all Operators of Library Base

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | BRANCH | Clones the input link to a number of output links you can define. | Version 1.1 |
|  | CastBitWidth | Changes the bit width by selecting the lower bits of the input. | Version 1.1 |
|  | CastColorSpace | Changes the logical attribute *ColorFlavor.* | Version 1.1 |
|  | CastKernel | This operator re-organizes the parallelism and kernel size of the incoming data. | Version 1.1 |
|  | CastParallel | Enables the re-interpretation of the input link bits. | Version 1.1 |
|  | CastType | Changes the link property Arithmetic | Version 1.1 |
|  | CONST | Replaces each input value by a fixed value at the output link. | Version 1.1 |
|  | ConvertPixelFormat | Preserve as much information as possible, while arithmetic and bit width are adjusted to a desired output pixel format. | Version 1.1 |
|  | Coordinate_X | Provides the x coordinate of the input pixel at its output. | Version 1.1 |
|  | Coordinate_Y | Provides the y coordinate of the input pixel at its output. | Version 1.1 |
|  | Dummy | Is a place holder for non existing operators. | Version 2 |
|  | DynamicROI | Allows an ROI selection using input links. | Version 1.2 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | EventToHost | Generates software events for rising edges at its input links. | Version 2.2 |
|  | EventDataToHost | This operator generates software events with data payload. | Version 3.5 |
|  | ExpandToKernel | The operator expands the input link I to an arbitrary kernel size. | Version 1.3 |
|  | ExpandToParallel | The operator expands the input link I with parallelism 1 to an arbitrary parallelism at output link O. | Version 1.1 |
|  | GetStatus | Obtain the current value of a signal link. | Version 1.2 |
|  | HierarchicalBox | Groups several operators into a single module. | Version 1.1 |
|  | ImageNumber | Provides the image index of the current image of the input link at the output link. | Version 1.1 |
|  | KernelRemap | Remapping of kernel components. | Version 1.4 |
|  | MergeComponents | Merges three gray input links to one RGB output link | Version 1.1 |
|  | MergeKernel | Merges the input links into a kernel. | Version 1.2 |
|  | MergeParallel | The operator merges N input links to a single output link by concatenating the input links to parallelism components. | Version 1.3 |
|  | MergePixel | Merges pixels of multiples input links of the same parallelism into one pixel of larger bit width. | Version 1.1 |
|  | NOP | No operation. Outputs the registered input. | Version 1.1 |
|  | PARALLELdn | Decreases the parallelism between the input link and the output link. | Version 1.1 |
|  | PARALLELup | Increases the parallelism between the input link and the output link. | Version 1.1 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | PseudoRandomNumberGen | Generates a stream of random values. | Version 1.4 |
|  | SampleDn | Reduces the image size by downsampling the input image. | Version 1.1 |
|  | SampleUp | Increases the image size by upsampling the input image. | Version 1.1 |
|  | SelectBitField | Enables the selection of a bit field. | Version 1.1 |
|  | SelectComponent | Extracts a single color component form the input link. | Version 1.1 |
|  | SelectFromParallel | Extracts a single parallelism component from the input link. | Version 1.1 |
|  | SelectROI | Extracts a rectangular region of interest (ROI) from the frames at the input link. | Version 1.1 |
|  | SelectSubKernel | Extracts a rectangular subset of the kernel matrix at the input link. | Version 1.1 |
|  | SetDimension | Enables to change of the link properties maximum image width and maximum image height. | Version 1.1 |
|  | SplitComponents | Separates the components of a color stream into three separate gray image streams. | Version 1.1 |
|  | SplitKernel | Splits the N x M kernel components of the input link into N * M output links. | Version 1.3 |
|  | SplitParallel | Splits the input link of parallelism degree N into N output links of parallelism degree 1 | Version 1.3 |
|  | Trash | A data sink for unused links. | Version 1.1 |

Table 20.1. Operators of Library Base

# 20.1. Operator BRANCH

**Operator Library: Base**

The *BRANCH* operator clones the input link to a parameterizable number of output links. You can define the number of output links when you add the operator to your design. No FPGA resources are required for this operator.



**Inserting a New *BRANCH* into an Existing Link**

Besides the possibility to insert a *BRANCH* operator form the operator library, you can also insert a *BRANCH* operator directly into an existing link. To do this, right click a link and select Split Link.... For more information, see Section 3.4.4, 'Inserting Operators' .

## 20.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O[0]..O[N-1], data output |

## 20.1.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O[0]..O[N-1] |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.1.3. Parameters

None

## 20.1.4. Examples of Use

The use of operator BRANCH is shown in the following examples:

- Section 3.6.2, 'O-Type Networks'

  Synchronization Rules - The use of the operator in an O-type Network.

# 20.2. Operator CastBitWidth

**Operator Library: Base**

The *CastBitWidth* operator changes the bit width by selecting the lower bits of the input. If the output bit width is greater than the input bit width, the value is kept, i.e. bits are added to the most significant bits. A sign extension is performed for signed value. If the output bit width is less than the input bit width, the most significant bits are discarded. The value changes to

$$OutputValue = InputValue \ \& \ (2^{w_o} - 1)$$

,i.e. only the remaining bits are used. The output bit width is defined by $w_o$ .

For color values, each component is processed individually.



**The *CastBitWidth* Operator Might Destroy Your Values**

Only discard bits, if you know you will not need them. The *CastBitWidth* operator might destroy your values.

| Input Bit Width | Input Value | | Arithmetic | Output Bit Width | Output Value | | Comment |
|---|---|---|---|---|---|---|---|
| | Decimal | Binary | | | Decimal | Binary | |
| 5 | 10 | 01010 | unsigned | 4 | 10 | 1010 | value is kept |
| 5 | 10 | 01010 | unsigned | 3 | 2 | 010 | value is changed |
| 5 | 10 | 01010 | signed | 6 | 10 | 001010 | value is kept |
| 5 | -10 | 10110 | signed | 6 | -10 | 110110 | value is kept, sign extension |
| 5 | 10 | 01010 | signed | 4 | -6 | 1010 | value is changed |
| 5 | -10 | 10110 | signed | 4 | 6 | 0110 | value is changed |

Table 20.2. Examples

## 20.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.2.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | [1, 64]❷ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

❷ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

### 20.2.3. Parameters

None

### 20.2.4. Examples of Use

The use of operator CastBitWidth is shown in the following examples:

- Figure 9.8, 'ShiftLeft Operator Added for 16Bit Output'

  Tutorial - User *ShiftLeft* to change DMA bit width.

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend'

  Tutorial - From equation to implementation. Explanation on how to implement the overlay blend.

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.2, 'Blob 2D'

  Examples - Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels.

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.4.4, 'RGB White Balancing'

  Examples - The applet shows an example for white balancing on RGB images.

- Section 12.8.1.2, 'Kirsch Filter'

  Examples - The Kirsch filter is a good edge detection filter for non directional edges.

- Section 12.8.1.4, 'Sobel Gradient X'

  Examples - A Sobel filter in x-direction only.

- Section 12.8.1.5, 'Sobel Multi Gradient'

  Examples - A Sobel filter in all 4 directions.

- Section 12.8.3.1, 'Averaging 3x3'

  Examples - A simple 3x3 box filter.

- Section 12.8.3.2, 'Gaussian Filter 5x5'

  Examples - A Gauss filter using a 5x5 kernel.

- Section 12.8.4.2, 'Parallel Filters'

  Examples - An example of the use of two filters in parallel.

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.9.6, 'Line Mirror'

  Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

- Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'

  Examples - High Dynamic Range According to Debevec

- Section 12.10.2, 'High Dynamic Range and Low Dynamic Range Example with a Weighted Linear Ansatz'

  Examples - High Dynamic Range with Linear Ansatz

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

# 20.3. Operator CastColorSpace

**Operator Library: Base**

The *CastColorSpace* operator changes the logical attribute *Color Flavor*. The values of the input link are copied to the output link. The operator doesn't influence the data stream and doesn't require any FPGA resources.

## 20.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.3.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [3, 63] unsigned, [6, 63] signed | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | as I |
| **Color Flavor** | any | any except FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 20.3.3. Parameters

None

## 20.3.4. Examples of Use

The use of operator CastColorSpace is shown in the following examples:

- Section 13.8, 'Functional Example for Specific Operators of Library Color, Base and Memory'

  Examples - Demonstration of how to use the operator

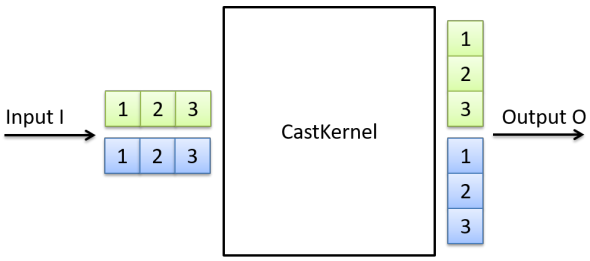
## 20.4. Operator CastKernel

**Operator Library: Base**

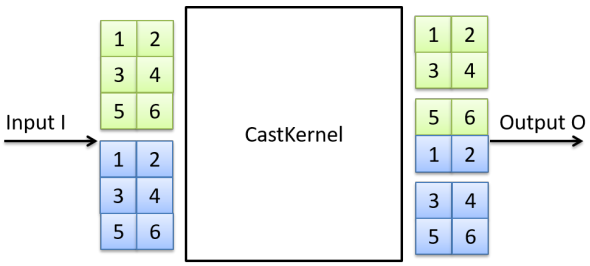
With the *CastKernel* operator, you can re-organize the parallelism and kernel size of the incoming data. For re-interpretation, change the kernel size at the output link. For example, an input link is configured with two kernel rows and two kernel columns and a parallelism of four. The *CastKernel* operator gives you the possibility to interpret these as four kernel rows and four kernel columns at parallelism 1 or as one kernel row and two kernel columns at parallelism 8, etc. The constraint of the *CastKernel* operator is that the product of kernel row and kernel column and parallelism must be identical for the input and the output link.

The following examples illustrate the conversion of kernel configuration and parallelism performed by the operator. The pseudo-code below the illustrations also describes the conversion pattern.

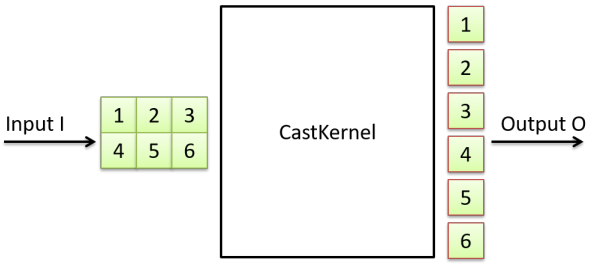
**Example 1: Converting the kernel size (col x row) 3x1 to the kernel size 1x3 while keeping parallelism 2:**




**Example 2: Converting parallelism 2 and kernel size (col x row) 2x3 to parallelism 3 and kernel size 2x2:**




**Example 3: Converting parallelism 1 and kernel size (col x row) 3x2 to parallelism 6 and kernel size 1x1:**




The operator changes the width of the images.

The mapping follows the following pseudo-code:

```
pi = 0
ri = 0
ci = 0
for p in 0 to P-1
  for r in 0 to R-1
    for c in 0 to C-1
      O[p][r][c] = I[pi][ri][ci]
```

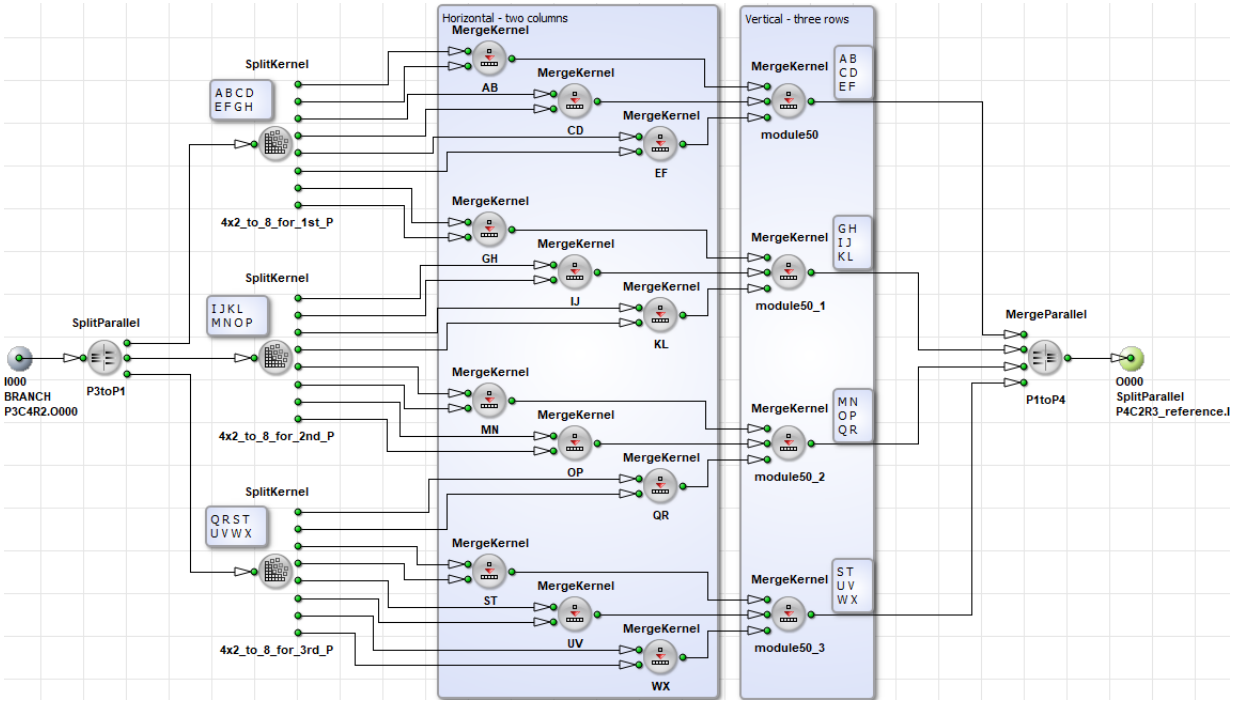
```
            ci = ci + 1
            if ci >= Ci then ci = 0, ri = ri + 1
            if ri >= Ri then ri = 0, pi = pi + 1
```

The pseudo-code has the following meaning:

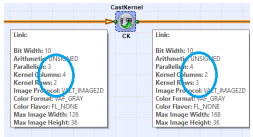
| Pseudo-code | Meaning |
|---|---|
| p | Output-Parallel-Index |
| pi | Input-Parallel-Index |
| r | Output-Kernel-Row-Index |
| ri | Input-Kernel-Row-Index |
| c | Output-Kernel-Column-Index |
| co | Output-Kernel-Column-Index |
| Pi | Input-Parallelism |
| P | Output-Parallelism |
| Ri | Input-Kernel-Rows |
| R | Input-Kernel-Columns |
| Ci | Input-KernelColumns |
| C | OutputKernelColumns |

Table 20.3. Explanation of pseudo-code

The function of the *CastKernel* operator could also be achieved by combining the follwoing operators: *SplitKernel*, *MergeKernel*, *SplitParallel* and *MergeParallel*:




In this example, the input configuration of four kernel columns and two kernel rows at parallelism 3 is re-organised to two kernel columns and three kernel rows at parallelism 4. The same can be achieved with the operator *CastKernel* by configuring the output link respectively:



### 20.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 20.4.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | As I |
| **Arithmetic** | {Unsigned, signed} | As I |
| **Parallelism** | Any | Auto❷ |
| **Kernel Columns** | Any | Any |
| **Kernel Rows** | Any | Any |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | As I |
| **Color Format** | Any | As I |
| **Color Flavor** | Any | As I |
| **Max. Img Width** | Any | Auto❸ |
| **Max. Img Height** | Any | As I |

❶ The range for bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63]

❷ The output parallelism $p_o$ is determined by the input parallelism $p_i$ , the input kernel size $r_i \times c_i$ and the output kernel size $r_o \times c_o$ :

$$p_o = p_i \times \frac{r_i \times c_i}{r_o \times c_o},$$

where $r$ denotes the kernel rows and $c$ denotes the kernel columns.

❸ The output maximum image width is determined by the input maximum image width, the output parallelism $p_o$ and the input parallelism $p_i$ by:

$$OutputMaxImgWidth = InputMaxImgWidth \times \frac{p_i}{p_o}$$

### 20.4.3. Parameters

None

### 20.4.4. Examples of Use

The use of operator CastKernel is shown in the following examples:

- Section 13.3, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

- Section 13.4, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

# 20.5. Operator CastParallel

**Operator Library: Base**

Library: Base

The operator CastParallel enables the re-interpretation of the input link bits. Change the parallelism at the output link for re-interpretation. For example, an input link width of 16-bit per pixel and a parallelism of two transmits 32-bits at each clock cycle. The CastParallel operator gives you the possibility to interpret these 32-bits as 8-bit per pixel at parallelism 4 or 4-bit per pixel at parallelism 8, etc. One constraint of the cast is that the product of the width and the parallelism must be identical for the input and the output link. The 2nd constraint is that the output pixel width must not exceed 64-bits.

At parallelism casts, the lower parallel components will be mapped to the lower bits.

Note that the operator will change the bit width of a pixel and the width of the images.

## 20.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.5.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | auto❷ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | any |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | auto❸ |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ The output bit width is determined from the output parallelism, the input parallelism and the input bit width by

$$w_o = w_i \times \frac{p_i}{p_o}$$

where $w_i$ = bit width at the input, $p_i$ = parallelism at the input, $p_o$ = output parallleism and $w_o$ = bit width at the output.

❸ The output image width must not exceed 2^31 - 1.

## 20.5.3. Parameters

None

### 20.5.4. Examples of Use

The use of operator CastParallel is shown in the following examples:

- Section 3.7.2.2, 'Parameter Editing'

  Design Parametrization - Invalid link property.

- Section 11.1, 'Functional Example for the *FrameBufferMultRoiDyn* Operator on the imaFlex CXP-12 Penta Platform'

  Examples - Demonstration of how to use the operator

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

- Section 12.9.6, 'Line Mirror'

  Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

# 20.6. Operator CastType

**Operator Library: Base**

The operator CastType changes the logical link property Arithmetic. The bits of the input and output values are equal. No changes of the value are done in hardware. Therefore, the operator does not require any FPGA resources.

| Bit Width | Input Value | | Input Arithmetic | Output Arithmetic | Output Value | | Comment |
|---|---|---|---|---|---|---|---|
| | Decimal | Binary | | | Decimal | Binary | |
| 4 | 5 | 0101 | unsigned | signed | 5 | 0101 | |
| 4 | 10 | 1010 | unsigned | signed | -6 | 1010 | interpretation of value has been changed |
| 4 | 5 | 0101 | signed | unsigned | 5 | 0101 | |
| 4 | -1 | 1111 | signed | unsigned | 15 | 1111 | interpretation of value has been changed |

Table 20.4. Examples

## 20.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.6.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.6.3. Parameters

None

### 20.6.4. Examples of Use

The use of operator CastType is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.9.6, 'Line Mirror'

  Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

- Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'

  Examples - High Dynamic Range According to Debevec

- Section 12.10.2, 'High Dynamic Range and Low Dynamic Range Example with a Weighted Linear Ansatz'

  Examples - High Dynamic Range with Linear Ansatz

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

# 20.7. Operator CONST

**Operator Library: Base**

The operator CONST provides a fixed value at the output link. All input values are discarded and replaced by the constant. The constant can be defined by parameter *Value*. This can either be a static or dynamic value. Thus the value can be changed during runtime.

Note that for multi component formats like RGB, the constant value is defined as a single value for the full pixel and not just the component.

Example: Output link RGB 24bit, constant value is set to 11862494 (0xB501DE). The output components will then be set to the following const values: R = 222 (0xDE), G = 01 (0x01) and B = 181 (0xB5). The constant concatenates the component values to a single value. The component 0 is placed in LSB area and the highest component in MSB area, i.e. for RGB, the constant value is {B-G-R}. Value = {component N-1, ..., component 0}, with N being the number of the components.

## 20.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.7.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | [1, 63]❷ |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | any❸ |
| **Kernel Rows** | any | any❹ |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | any |
| **Color Flavor** | any | any |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The range of the output for unsigned arithmetic is [1, 63]. For signed outputs, the range is [2, 64]. For unsigned color outputs, the range is [3, 63] and for signed color, the range is [6, 63].
❸❹ The same const value is used for all kernel components.

## 20.7.3. Parameters

| Value | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | depends on arithmetic and bit width of the output link |

| Value |
| --- |
| This parameter defines the value at the output link. If the parameter is set to Static, the value is selected at design time. If the parameter is Dynamic, it is possible to alter the value during runtime. |

### 20.7.4. Examples of Use

The use of operator CONST is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.2.1, 'Adaptive Threshold'

  A binarization example for local adaptive thresholding. A kernel size of 8 by 8 pixel is used.

- Section 12.9.6, 'Line Mirror'

  Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

# 20.8. Operator ConvertPixelFormat

**Operator Library: Base**

The operator ConvertPixelFormat tries to preserve as much information as possible, while arithmetic and bit width are adjusted to a desired output pixel format. Thus the operator allows the definition of the bit width and arithmetic at the output link.

The main principle of algorithm is twofold:

1. Change the arithmetic by clipping
   - unsigned to signed: the MSB at the output is used as a sign bit
   - signed to unsigned: clipping to positive values i.e. if a value is less than 0 it is set to zero
2. Change the bit width y keeping the relative brightness
   - if the output bit width is increased, the value is left shifted to meet the desired bit width
   - if the output bit width is decreased, the value is divided and rounded to meet the desired bit width

Thus the output value can be determined by:

$$s_i = \begin{cases} 1 & \text{if input arithmetic} = signed \\ 0 & \text{otherwise} \end{cases}$$

$$s_o = \begin{cases} 1 & \text{if output arithmetic} = signed \\ 0 & \text{otherwise} \end{cases}$$

$$w_i = \text{input bit width}$$

$$w_o = \text{output bit width}$$

$$OutputValue = \begin{cases} 2^{w_o - s_o} & \text{if } InputValue \times 2^{w_o - w_i - s_o + s_i} > 2^{w_o - s_o} - 1 \\ 0 & \text{if } s_o = 1 \ \& \ InputValue < 0 \\ Round\left(InputValue \times 2^{w_o - w_i - s_o + s_i}\right) & \text{otherwise} \end{cases}$$

| Input Bit Width | Output Bit Width | Input Arithmeti | Output Arithmeti | Input Value | | Output Value | | Comment |
|---|---|---|---|---|---|---|---|---|
| | | | | Decimal | Binary | Decimal | Binary | |
| 4 | 5 | unsigned | unsigned | 10 | 1010 | 20 | 10100 | value * 2 |
| 5 | 4 | unsigned | unsigned | 15 | 1111 | 8 | 1000 | value divided by 2 and rounded |
| 5 | 4 | unsigned | unsigned | 31 | 11111 | 15 | 1111 | value clipped to maximum |
| 4 | 4 | unsigned | signed | 11 | 1011 | 6 | 0100 | value divided by 2 and rounded |
| 4 | 4 | signed | unsigned | 4 | 0100 | 8 | 1000 | value * 2 |
| 5 | 4 | signed | unsigned | 10 | 01010 | 10 | 1010 | value kept |
| 5 | 4 | signed | unsigned | -2 | 11110 | 0 | 0000 | value clipped to 0 |
| 5 | 4 | signed | signed | -7 | 11001 | -4 | 1100 | value divided |

| Input Bit Width | Output Bit Width | Input Arithmeti | Output Arithmeti | Input Value | | Output Value | | Comment |
|---|---|---|---|---|---|---|---|---|
| | | | | Decimal | Binary | Decimal | Binary | |
| | | | | | | | | by 2 and rounded |

Table 20.5. Examples

For color links, each color component is processed individually.

### 20.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 20.8.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | [1, 64]❷ |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The range of the output bit width is [1, 64]. For signed outputs, the range is [2, 64]. For unsigned color outputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 20.8.3. Parameters

None

### 20.8.4. Examples of Use

The use of operator ConvertPixelFormat is shown in the following examples:

- Figure 9.7, 'ConvertPixelFormat Operator Added for 16Bit Output'

  Tutorial - User *ConvertPiyelFormat* to change DMA bit width.

- Section 11.8, 'Example for the *DMAFromPC* Operator on the imaFlex CXP-12 Quad Platform'

  Examples - Demonstration of how to use the operator using the example of shading correction

- Section 12.4.1.2, 'Bayer 3x3 De-Mosaicing'

Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter.

- Section 12.4.1.3, 'Bayer 5x5 De-Mosaicing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter.

- Section 12.4.1.4, 'Bayer 3x3 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.9, 'Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern Red/BlueFollowedByGreen GreenFollowedByBlue/Red '

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a bilinear line scan camera with color pattern Red/BlueFollowedByGreen_GreenFollowedByBlue/Red

- Section 12.4.1.10, 'Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern RedFollowedByBlue GreenFollowedByGreen '

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a bilinear line scan camera with color pattern Red/BlueFollowedByBlue/Red_GreenFollowedByGreen

- Section 12.4.1.11, 'Bayer DeMosaicing a Line Scan Camera with Bi-color Bayer Pattern biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a CXP-12 line scan camera with bi-color Bayer pattern: biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR, for example the racer 2 L camera. In addition, the example contains a line scan trigger module and a white balancing module.

- Section 12.5.2, 'Co-Processor Large Filter Calculation'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a large filter kernel is calculated.

- Section 12.7.2, 'Noise Reduction'

  Examples - The average of two acquired images is calculated to reduce noise.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

# 20.9. Operator Coordinate_X

**Operator Library: Base**

The operator Coordinate_X provides the x coordinate of the input pixels at its output. Thus, each pixel at the input is replaced by its x-position in the image. The actual input pixel values are not used. The operator works as a counter which is incremented with every new pixel and is reset after the end of a line. The first pixel of each row will have coordinate zero.

## 20.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.9.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | auto❷ |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ The output bit width is automatically determined from the maximum image width i.e.

$$OutputBitWidth = log_2(MaxImageWidth)$$

## 20.9.3. Parameters

None

## 20.9.4. Examples of Use

The use of operator Coordinate_X is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

- Section 12.9.6, 'Line Mirror'

  Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

# 20.10. Operator Coordinate_Y

**Operator Library: Base**

The operator Coordinate_Y provides the y coordinate of the input pixels at its output. Thus, each pixel at the input is replaced by its y-position in the image. The actual input pixel values are not used. The operator works as a counter which is incremented with every new line and is reset after the end of a frame. The pixels of the first image row will have coordinate zero.

## 20.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.10.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | auto❷ |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ The output bit width is automatically determined from the maximum image height i.e.

$$OutputBitWidth = log_2(MaxImageHeight)$$

## 20.10.3. Parameters

None

## 20.10.4. Examples of Use

The use of operator Coordinate_Y is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

- Section 12.9.4, 'ImageSplitAndMerge'

Examples - Shows how to split an merge image streams. Appends a trailer to the image.

# 20.11. Operator Dummy

**Operator Library: Base**

The *Dummy* operator is automatically inserted into the design if the required operator is not available. This can have one of the following reasons:

- The VisualApplet project has been ported from another hardware platform. If the operator is not available for the new platform, the *Dummy* operator is inserted. Read Section 4.5, 'Target Hardware Porting' for more information.
- The project was originally generated in an older version of VisualApplets. In the current version, the operator might not be available anymore. Check the release notes of the current version and Section 4.14, 'Migration from Older Versions' for more information. Most operators can be replaced by other operators with similar names.
- The VisualApplets installation is incomplete. Some files might be missing. Reinstall VisualApplets into an empty directory.

To inform you about the module name and the parameter values of the original operator, check the parameters.

The number of links differs. They replace the old paramerer links.



**Dummy Operator has to be Replaced**

A VisualApplets project cannot be build with a *Dummy* operator inside. All *Dummy* operators have to be replaced or removed.

## 20.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | |

## 20.11.2. Supported Link Format

None

## 20.11.3. Parameters

| Replaces | |
|---|---|
| **Type** | static parameter |
| **Default** | |
| **Range** | |
| Contains the original operator name. | |

| Varius Parameter Names | |
|---|---|
| **Type** | static parameter |
| **Default** | |
| **Range** | |
| For each parameter of the original operator, a parameter exists. The name and the value is equal to the original. | |

# 20.12. Operator DynamicROI

**Operator Library: Base**

The operator DynamicROI extracts a rectangular region of interest (ROI) from the input link I. The output image height and line width are defined by the ROI image parameters Ylength and Xlength. The size of this ROI is defined dynamically by the four input links Xoffset, Yoffset, Xlength, and Ylength. The purpose of the operator is to adapt the ROI dimension and location to results of image processing steps. All parameters of the ROI may vary from image to image.

The image stream from I to O is controlled by the four links Xoffset, Yoffset, Xlength, and Ylength. These four links can work fully asynchronous (sourced by different M type operators) as well as fully synchronous (sourced by the same M type operator through an arbitrary network of O type operators). To output an ROI at output O it is required to provide an image at all five inputs. Those inputs where an image is already provided are blocked until at all inputs an image is provided. Buffering elements at the indivdual inputs, especially at input link I, are highly recommended. You can imagine the behavior of DynamicROI as a valve which only opens if an image is provided at all inputs.

Whenever an image arrives at each of the four control inputs the very last pixel of each control link is captured by the DynamicROI. As soon as DynamicROI received all images at the control inputs, the valve opens and an image is allowed to pass from I to O. Now the captured ROI coordinates are used to extract the ROI. After the ROI image has been transferred to O, the valve closes again and waits for the next image ROI coordinate set appearing at the control links.

An important use of this operator is in conjunction with CreateBlankImage and CoefficientBuffer or with the results of an object detection implementation. An operator to control the ROI coordinates using parameter is *SelectROI*.

**Operator Restrictions**

- If the Xlength is not an integer multiple of the parallelism of links I and O, the operator will extend the ROI width to match with the parallelism. Dummy pixels are added to the end of each line. The value of that dummy pixel is undefined. In VA simulation dummy pixels will be set to zero for better visibility.
- If the ROI coordinates are illegal, i.e. the Xlength and/or Ylength are zero or the offsets exceed the image boundaries, the operator will cut and output an empty image or will remove the image. The behavior can be controlled using parameter *SuppressEmptyRoI*.
- If the offset coordinates are valid but the Xlength and/or Ylength coordinates exceed the image boundaries, the operator will shrink ROI to the image boundaries.
- When an empty coordinate set is provided on ROI inputs, i.e. a ROI frame of height and width equal to 0, operator DynamicROI treats such an empty ROI as a ROI with invalid coordinates. The operator will cut and output an empty image or will remove the image. The behavior can be controlled using parameter *SuppressEmptyRoI*.
- Empty images on port I are not supported.
- The lines of each input image at port I must have the same length. Thus images with varying line lengths are not allowed.

## 20.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image input<br>Xoffset, x-offset input<br>Yoffset, y-offset input<br>Xlength, x-length input<br>Ylength, y-length input |
| **Output Link** | O, data output |

**Synchronous and Asynchronous Inputs**

- All inputs are asynchronous, i.e. they may be sourced by different M-type operators.

## 20.12.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Xoffset | Input Link Yoffset |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | auto❷ | auto❸ |
| **Arithmetic** | {unsigned, signed} | unsigned | unsigned |
| **Parallelism** | any | any | any |
| **Kernel Columns** | any | any | any |
| **Kernel Rows** | any | any | any |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | VALT_IMAGE2D |
| **Color Format** | any | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | any | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any | any |
| **Max. Img Height** | any | any | any |

| Link Parameter | Input Link Xlength | Input Link Ylength | Output Link O |
|---|---|---|---|
| **Bit Width** | auto❹ | auto❺ | as I |
| **Arithmetic** | unsigned | unsigned | as I |
| **Parallelism** | any | any | as I |
| **Kernel Columns** | any | any | as I |
| **Kernel Rows** | any | any | as I |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | as I |
| **Color Format** | VAF_GRAY | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | FL_NONE | as I |
| **Max. Img Width** | any | any | as I |
| **Max. Img Height** | any | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷❸❹❺ The allowed bit width at the control inputs is automatically determined from the maximum image dimensions of the input link I

$$BitWidth = Max(Ceil(log2(MaxImageWidth + 1)), Ceil(log2(MaxImageHeight + 1))$$

## 20.12.3. Parameters

| SuppressEmptyRoI | |
|---|---|
| **Type** | static parameter |
| **Default** | ON |
| **Range** | {ON, OFF} |
| This parameter controls the removal of empty images on the operator's output if the ROI coordinates are illegal. See Operator Restrictions for more information. | |

## 20.12.4. Examples of Use

The use of operator DynamicROI is shown in the following examples:

- Section 12.3.3, 'Blob2D ROI Selection'

Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

## 20.13. Operator EventToHost

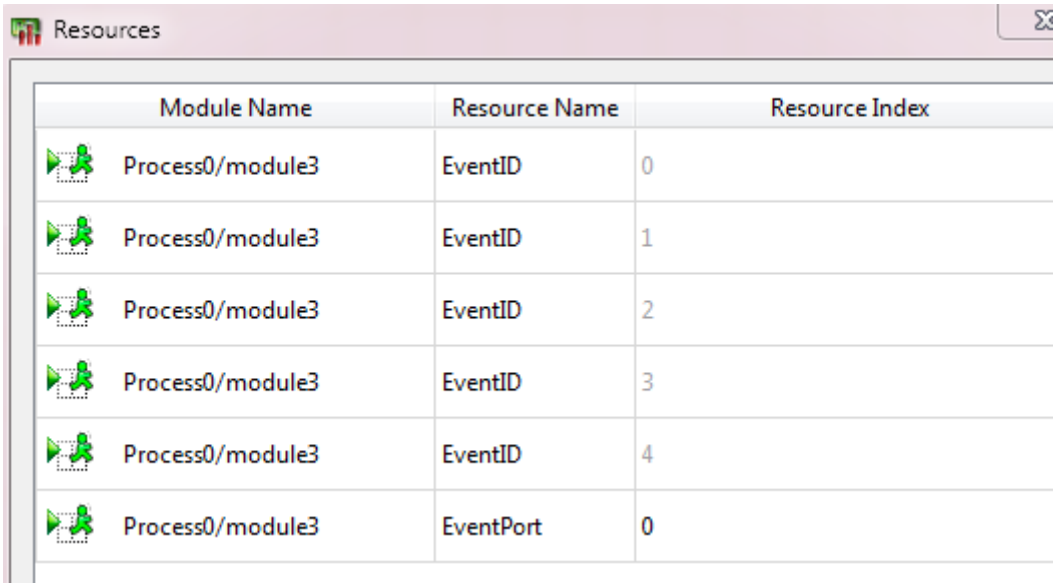
**Operator Library: Base**

The operator generates software events for rising edges at its input links. The operator provides up to 16 input ports which can raise events.

All N inputs (up to 16) can generate individual events. The event will not provide any signal link data. Commonly, this operator is used to monitor the status of GPIOs or to signal special conditions.




Each input port is associated with the correspondent event. The event is identified by the software application by using the unique event name (parameter **EventName_** (see parameter description below)).




The operator uses one resource of type EventPort exclusively. You can modify resource EventPort. EventPort specifies which event chanel is used by the software. Each EventPort number can only be used once in a design.

In addition, for each input port a resource of type EventID is reserved. You cannot modify this resource (therefore displayed in grey) as the EventID is generated automatically. If, via copy & paste, you have the same EventID in different operator instances, you need to delete one of these instances and to instanciate the operator anew.



**Limited Amount of Event Ports and of Individual Events**

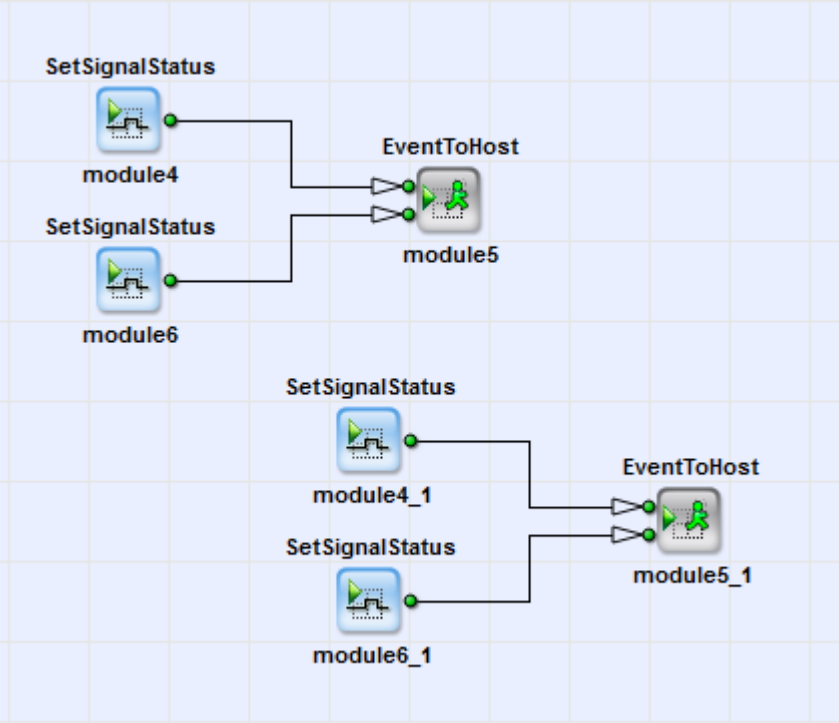
The maximum amount of event ports available in a design depends on the target hardware platform you are designing for (see Apendix, section Section A.2, ' Device Resources of Supported Platforms ' for detailed information).

If you design an applet for use in the runtime environment, you can use a maximum of 64 individual events in a design.



### After Copy & and Pasting the Operator

After Copy & and Paste, you have to adapt the resources to ensure each EventPort and each EvendID is used only once in the design.

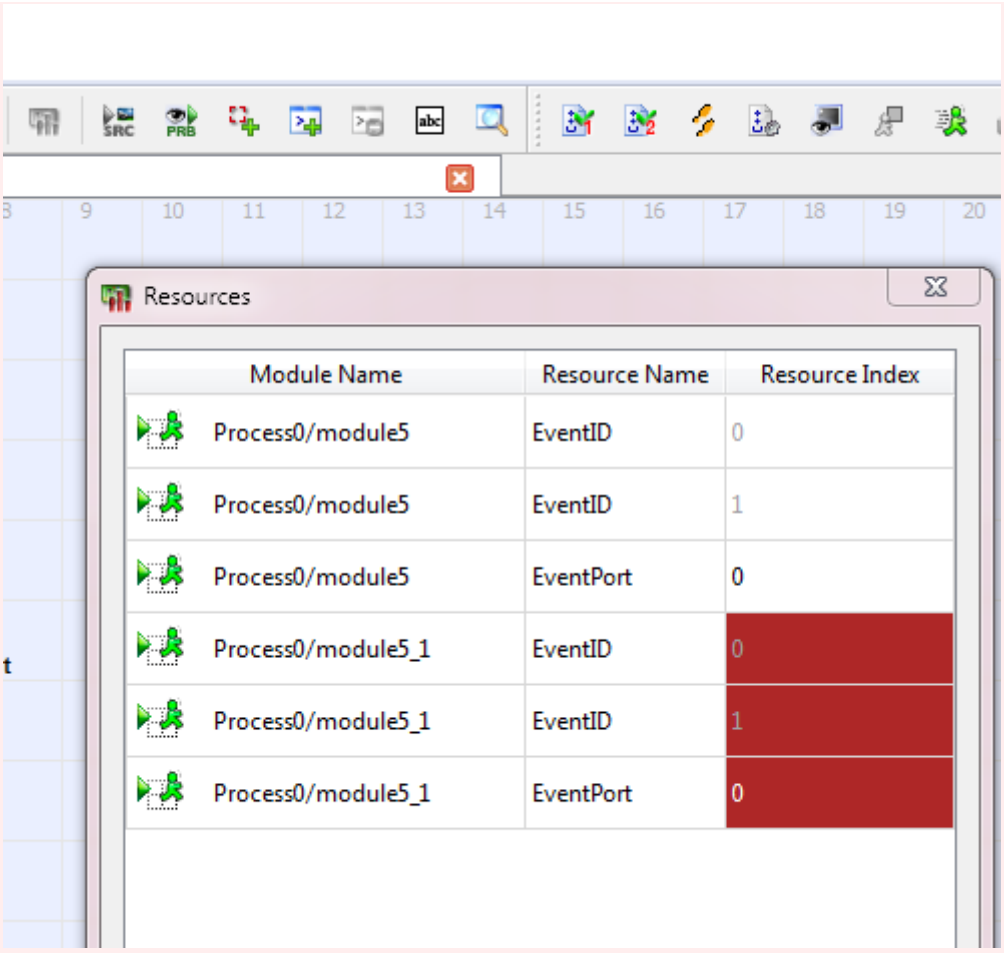



To adapt the resources:

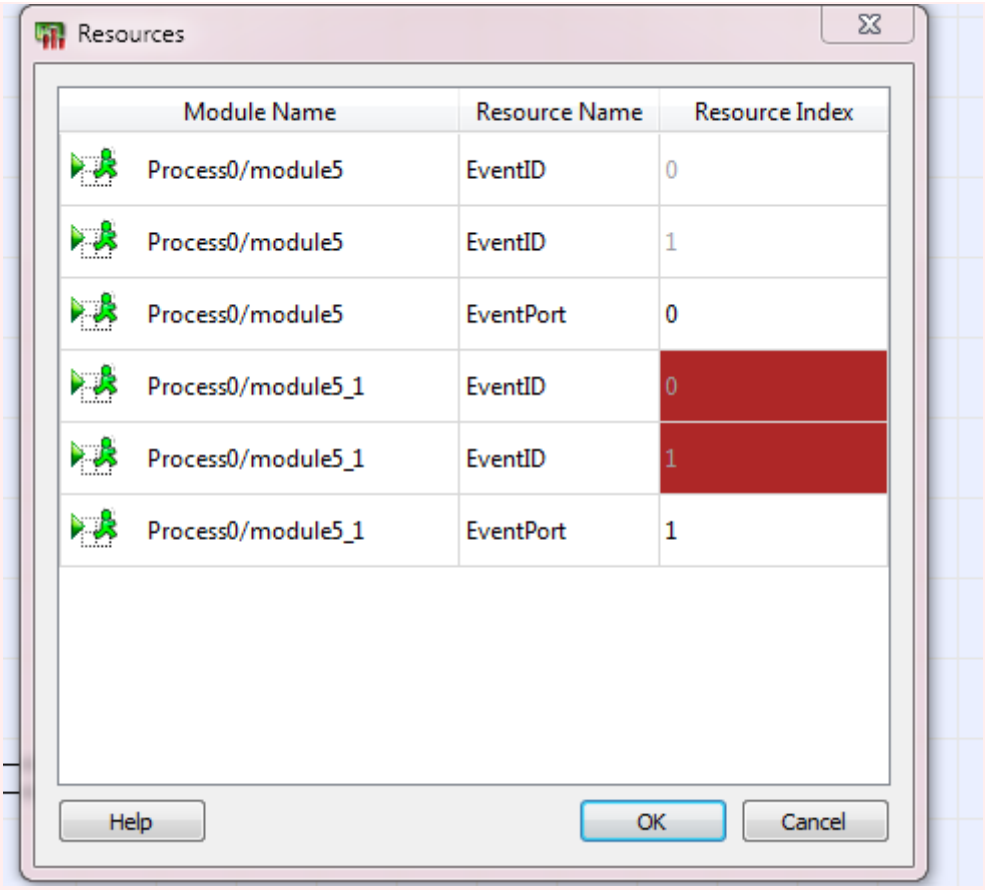
1. Click on the Resources button in the tool bar.



The resources that are overmapped are displayed in red:

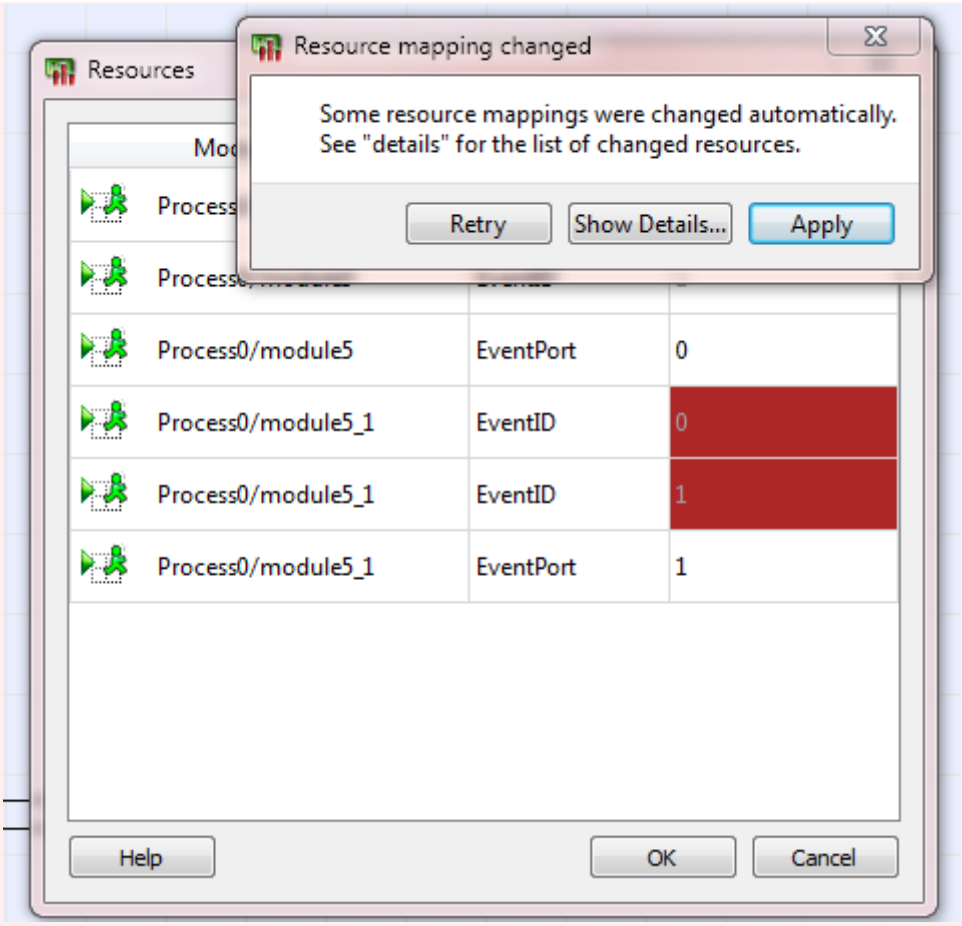



2. Enter a EventPort number for the event port that has not been used so far in the design:

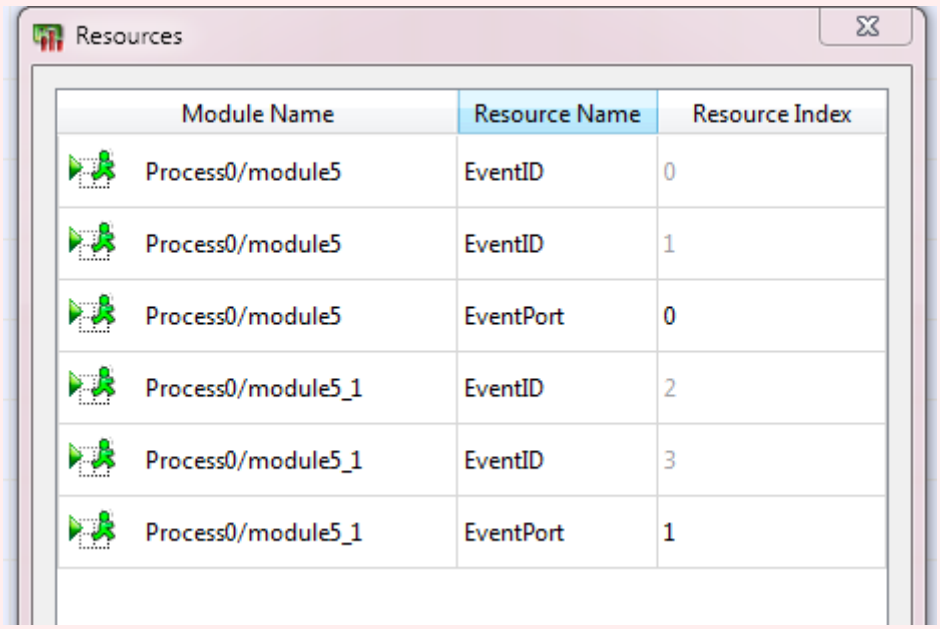



Since the EventID is not editable, but generated automatically, you cannot enter new values for EventID.

3. Click the **OK** button. The following message is displayed:




4. Click on **Apply**. The Resources dialog closes. If you re-open it, you will see, that for the EventIDs unique values have been generated:



### 20.13.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I[000-015], Signal input to raise a software event. |

### 20.13.2. Supported Link Format

| Link Parameter | Input Link I[000-015] |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 20.13.3. Parameters

| **EventsWithTimestamp** | |
|---|---|
| **Type** | static/dymanic write parameter |
| **Default** | ON |
| **Range** | {ON,OFF} |
| Defines if high-precision timestamps are attached to each event.<br>ON = timestamps are generated.<br>OFF = no timestamps are generated. | |

| **EventName_[n]** | |
|---|---|
| **Type** | static write parameter |
| **Default** | EventName_[n] |
| **Range** | |
| Every event input must be assigned a unique identifier name. This event name is used to identify and use a particular hardware event signal in the Framegrabber SDK. | |

### 20.13.4. Examples of Use

The use of operator EventToHost is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

# 20.14. Operator EventDataToHost

**Operator Library: Base**

The operator generates software events with data payload. The payload size can be configured as N 16-bit data words via the *PayloadSize* parameter.

The operator has a 16-bit data input I and a 1-bit strobe input S. The inputs are O-synchronous. The input S starts a new event. Any data which arrives at I before S is asserted is ignored. S may be asserted for more than one data cycle but once S is de-asserted it may not be asserted again before the payload data for the ongoing event is completed. If an automatic clearance is set (see parameter *AutoClear*), then S can even be constantly 1.

The parameter *AutoClear* defines at which time clearance is done so that another event can be triggered:

- EoL: Another event may be started after the next end of line. As soon as the N-th data is ready, no further event data is accepted until the end of the line, independent of the state of S. Starting with the next line, a new event may be generated with S=1.
- EoF: Another event may be started after the next end of frame. As soon as the N-th data is ready, no further event data is accepted until the end of the frame, independent of the state of S. Starting with the next frame, a new event may be generated with S=1.
- NONE: No auto-clearance. S must transfer a 0 for at least one data cycle after the event strobe to activate that event data for a subsequent event is accepted. Note that once S is de-asserted, it may not be asserted again before the payload data for the ongoing event is completed.

The operator stores the data payload and triggers the event as soon as N words of data have been received from I. If the event data is incomplete at the time when auto clearance happens, no event is generated and the incomplete payload data is deleted.

The input of the operator is never blocked. As the capacity of the event transfer buffers is limited, overflow may occur when the event rate is too high. To detect data loss, it may be useful to include a sequential number in your event data.

The triggered event is identified by the software application by using the unique event name via *EventName* parameter (see parameter description below).

The operator uses one resource of type *EventPort* exclusively. You can modify the resource *EventPort*. *EventPort* specifies which event channel is used by the software. Each *EventPort* number can only be used once in a design.

In addition, one resource of type *EventID* is reserved. You can't modify this resource (therefore it is greyed out), because the *EventID* is generated automatically. If you copy & paste an operator instance, and thus have the same *EventID* in different operator instances, you must delete one of these instances and instantiate the operator anew.



**Limited Amount of Event Ports and of Individual Events**

The maximum amount of event ports available in a design depends on the target hardware platform you are designing for (see Section A.2, ' Device Resources of Supported Platforms ' for detailed information).

If you design an applet for use in the runtime environment, you can use a maximum of 64 individual events in a design.

## 20.14.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, Input for payload data. |

| Property | Value |
|---|---|
| | S, Strobe for starting new event. |

### 20.14.2. Supported Link Format

| Link Parameter | Input Link I | Input Link S |
|---|---|---|
| **Bit Width** | 16 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 20.14.3. Parameters

| **EventsWithTimestamp** | |
|---|---|
| **Type** | static/dymanic write parameter |
| **Default** | ON |
| **Range** | {ON,OFF} |
| Defines whether high-precision timestamps are attached to each event.<br>ON: Timestamps are generated.<br>OFF: No timestamps are generated. | |

| **EventName** | |
|---|---|
| **Type** | static write parameter |
| **Default** | Event |
| **Range** | |
| This event name is used to identify and use a particular event signal in the Framegrabber SDK. | |

| **PayloadSize** | |
|---|---|
| **Type** | static parameter |
| **Default** | 1 |
| **Range** | [1, 254] |
| Set the payload size as a number of 16-bit data words. | |

| **AutoClear** | |
|---|---|
| **Type** | static parameter |
| **Default** | NONE |
| **Range** | {NONE, EoL, EoF} |
| Defines at which time clearance is done so another event can be triggered.<br><br>When EoL is set, then after the next end of line another event may be started. As soon as the N-th data is ready, no further event data is accepted until the end of the line, independent of the state of S. Starting with the next line, a new event may be generated with S=1. | |

| **AutoClear** |
| --- |
| When EoF is set, then after the next end of frame another event may be started. As soon as the N-th data is ready, no further event data is accepted until the end of the frame, independent of the state of S. Starting with the next frame, a new event may be generated with S=1.<br><br>When NONE is set, then S must transfer a 0 for at least one data cycle after the event strobe to activate that event data for a subsequent event is accepted. Note that once S is de-asserted, it may not be asserted again before the payload data for the ongoing event is completed. |

# 20.15. Operator ExpandToKernel

**Operator Library: Base**

The operator expands the input link I to an arbitrary kernel size. The input value is replicated/copied to each of the kernel components. Set the new kernel size at the output link.

## 20.15.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.15.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | any |
| **Kernel Rows** | 1 | any |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.15.3. Parameters

None

## 20.15.4. Examples of Use

The use of operator ExpandToKernel is shown in the following examples:

- Section 12.13.2, 'Print Inspection Example- Position Correction and Defect Detection Using Blob Based Template Matching'

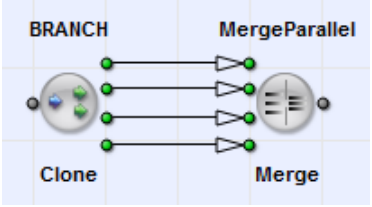
  Examples- Geometric Transformation and Defect Detection

# 20.16. Operator ExpandToParallel

**Operator Library: Base**

The operator expands the input link I with parallelism 1 to an arbitrary parallelism at output link O. The input value is replicated/copied to each parallel stream. Set the new parallelism at the output link.

The performed operation can be described as combination of the existing operators *BRANCH* and *MergeParallel*:




In this example, an input link with parallism 1 is cloned to parallelism 4.

## 20.16.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.16.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | As I |
| **Arithmetic** | {Unsigned, signed} | As I |
| **Parallelism** | 1 | Any |
| **Kernel Columns** | Any | As I |
| **Kernel Rows** | Any | As I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | As I |
| **Color Format** | Any | As I |
| **Color Flavor** | Any | As I |
| **Max. Img Width** | Any | Auto❷ |
| **Max. Img Height** | Any | As I |

❶ The range of the input bit width is:

- For unsigned values: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63]

❷ The output maximum image width is determined by the input maximum image width and the output parallelism $p_o$ :

$$OutputMaxImgWidth = InputMaxImgWidth \times p_o$$

### 20.16.3. Parameters

None

### 20.16.4. Examples of Use

The use of operator ExpandToParallel is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

  Examples - Demonstration of how to use the operator

# 20.17. Operator GetStatus

**Operator Library: Base**

The operator monitors the state of the input pixel stream and provides a register for reading out of the latest valid value via software.

## 20.17.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |

## 20.17.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed |
| **Arithmetic** | {unsigned, signed} |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 20.17.3. Parameters

| Status | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | InputBitWidth |
| The latest valid value from the input pixel stream is stored in the register Status which is readable via software.<br><br>Note that the parameter is always of type unsigned. If you connected a signed link to the operator, reinterpret the value as signed value in your software. | |

## 20.17.4. Examples of Use

The use of operator GetStatus is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

# 20.18. Operator HierarchicalBox

**Operator Library: Base**

The HierarchicalBox is a pseudo module provided by the graphical user interface to group several operators into a single module. This makes it feasible for users to maintain complex projects and comfortably re-use parts from previous projects.

On instantiation of a HierachicalBox, it is possible to select an arbitrary number of input and output links. To open a view of the inside, double click onto the HieracialBox.

A detailed explanation of the usage of hierarchical boxes can be found in the user manual of VisualApplets in Section 4.1, 'Hierarchical Boxes'.

## 20.18.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | Operator type depends on context |
| **Input Link** | I[0] .. I[63], data input |
| **Output Link** | O[0] .. O[63], data output |

## 20.18.2. Supported Link Format

| Link Parameter | Input Link I[0] .. I[63] | Output Link O[0] .. O[63] |
|---|---|---|
| **Bit Width** | any | any |
| **Arithmetic** | any | any |
| **Parallelism** | any | any |
| **Kernel Columns** | any | any |
| **Kernel Rows** | any | any |
| **Img Protocol** | any | any |
| **Color Format** | any | any |
| **Color Flavor** | any | any |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

## 20.18.3. Parameters

None

## 20.18.4. Examples of Use

The use of operator HierarchicalBox is shown in the following examples:

- Section 4.1, 'Hierarchical Boxes'

  Usage and Navigation through design with *HierarchicalBox* modules in VisualApplets.

- Section 10.1.1.4, 'RGB Camera Link Medium Area'

  Tutorial - Basic Acquisition

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

# 20.19. Operator ImageNumber

**Operator Library: Base**

The operator ImageNumber provides the index of the current image of the input link at the output link. Thus, each pixel at the input is replaced by the index of the image i.e. the image number. The actual input pixel values are not used. The operator works as a counter which is incremented with every new image at its input. The counter is reset on the start of the process (Section 4.4, 'Multiple Processes') or after it reached it's maximum. The first image will have image index 0.

The counter width can be defined by changing the bit width of the output link O. If the counter reached it's maximum it will start from zero again = wrap around condition. For example, an output bit width of 8 bit will result in counter values between zero to 255.

Using parameter *SingleShot*, the operator will not be reset on the wrap around condition. The counter value output will remain at the maximum value. Thus, for an 8 bit image counter the counter value will remain at it's maximum of 255.

Often this operator is used in conjunction with operator *RemoveImage* to remove the first N images of a sequence.

**Operator Restrictions**

- The operator supports empty images. The operator will count those empty images. However, for the empty images, the counter value cannot be provided at the output as the output of an empty frame contains no pixel. After the next non-empty image the output link will provide the correct counter value.

## 20.19.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.19.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | [1, 64] |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 20.19.3. Parameters

| SingleShot | |
|---|---|
| **Type** | static parameter |
| **Default** | OFF |
| **Range** | {OFF, ON} |
| This parameter controls the wrap around condition. When set to OFF the counter will count infinitely with each processed image, i.e. a wrap around will occur at maximum possible counter value defined by the output link bit width.<br><br>When set to ON, the counter keeps counting until the maximal possible value is reached. The counter will stop counting at it's maximum value. | |

### 20.19.4. Examples of Use

The use of operator ImageNumber is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

- Section 12.6.4, 'Manual Image Injection'

  Example - For debugging purposes images can be inserted manually.

- Section 12.6.5, 'Image Monitoring'

  Example - For debugging purposes image transfer states on links can be investigated.

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

# 20.20. Operator KernelRemap

**Operator Library: Base**

The operator KernelRemap allows the remapping of kernel components between the input and the output. The output of the operator has the same kernel dimension as the input. Using the static parameter *SourceSelect*, it is possible to remap the kernel components. This can be useful to mirror a kernel or it can be useful for a rotation of the kernel components.

The remapping is done by allocating a new input kernel index to each output. Hence, for each output kernel component an input kernel component is selected using the parameter SourceSelect. It is also possible to allocate the same input kernel component to multiple output kernel components. If an input kernel component is not mapped to any output, the data is discarded.

## 20.20.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.20.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.20.3. Parameters

| SourceSelect | |
|---|---|
| **Type** | static parameter |
| **Default** | index |
| **Range** | [0, KernelRows * KernelCols - 1] |
| This parameter defines the remapping of the kernel. It represents the output kernel components. Hence, for every output kernel component an input kernel component is selected.<br><br>The kernel components are indexed from top-left to bottom-right. | |

### 20.20.4. Examples of Use

The use of operator KernelRemap is shown in the following examples:

- Section 12.9.7, 'Shear of an Image'

  Example - Line Shear example with linear interpolation.

# 20.21. Operator MergeComponents

**Operator Library: Base**

The operator MergeComponents merges three gray input links to one color output link. The opposite of this operator is Section 20.37, 'SplitComponents'.

## 20.21.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input color component {R, H, Y, L, X}<br>I1, data input color component {G, S, U, A, Y}<br>I2, data input color component {B, I, V, B, Z} |
| **Output Link** | O, data output |

## 20.21.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1 |
|---|---|---|
| **Bit Width** | [1, 21] unsigned, [2, 21] signed | as I0 |
| **Arithmetic** | {unsigned, signed} | as I0 |
| **Parallelism** | any | as I0 |
| **Kernel Columns** | any | as I0 |
| **Kernel Rows** | any | as I0 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I0 |
| **Color Format** | VAF_GRAY | as I0 |
| **Color Flavor** | FL_NONE | as I0 |
| **Max. Img Width** | any | as I0 |
| **Max. Img Height** | any | as I0 |

| Link Parameter | Input Link I2 | Output Link O |
|---|---|---|
| **Bit Width** | as I0 | auto❶ |
| **Arithmetic** | as I0 | as I |
| **Parallelism** | as I0 | as I |
| **Kernel Columns** | as I0 | as I |
| **Kernel Rows** | as I0 | as I |
| **Img Protocol** | as I0 | as I |
| **Color Format** | as I0 | VAF_COLOR |
| **Color Flavor** | as I0 | any |
| **Max. Img Width** | as I0 | as I |
| **Max. Img Height** | as I0 | as I |

❶ The output bit width is the sum of the three input bit widths i.e. I0 + I1 + I2.

## 20.21.3. Parameters

None

### 20.21.4. Examples of Use

The use of operator MergeComponents is shown in the following examples:

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend'

  Tutorial - Split and Merge Color components.

- Section 10.1.1.4, 'RGB Camera Link Medium Area'

  Tutorial - Basic Acquisition

- Section 11.6, 'Functional Example for the *JPEG_Encoder_Color_iF* User Library Element on the imaFlex CXP-12 Quad Platform'

  Examples - Demonstration of how to use the operator

- Section 11.7, 'Functional Example for the *JPEG_Encoder_Color_iF_Penta* User Library Element on the imaFlex CXP-12 Penta Platform'

  Examples - Demonstration of how to use the operator

- Section 12.1.6, 'Laser Pointer Detection'

  Examples - A convolution with high intensity spot coefficients is made. For results above threshold, the respective pixels are dyed in red.

- Section 12.4.1.4, 'Bayer 3x3 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.6, 'Edge Sensitive Bayer De-Mosaicing Algorithm'

  Examples - Edge Sensitive Laplace Bayer De-Mosaicing filter

- Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche'

  Examples - Laroche Bayer De-Mosaicing filter

- Section 12.4.1.8, 'Modified Laroche Bayer De-Mosaicing Algorithm '

  Examples - Ressource Optimized Laroche Bayer De-Mosaicing filter

- Section 12.4.4, 'RGB White Balancing'

  Examples - The applet shows an example for white balancing on RGB images.

- Section 12.6.4, 'Manual Image Injection'

  Example - For debugging purposes images can be inserted manually.

- Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'

  Examples - High Dynamic Range According to Debevec

- Section 12.10.2, 'High Dynamic Range and Low Dynamic Range Example with a Weighted Linear Ansatz'

  Examples - High Dynamic Range with Linear Ansatz

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

Examples - Rolling Average - Loop

# 20.22. Operator MergeKernel

**Operator Library: Base**

The operator MergeKernel merges an arbitrary number N of input links I[0] .. I[63] (max.) into a single output link O.

The operator allows kernels at its inputs. One kernel dimension (row or column) needs to have the same size (number of pixels) at all inputs. For the other dimension (rows or columns) an individual size can be defined for each input. Thus, either the number of rows or the number of columns has to be the same for all kernels at the inputs.

For detailed information, see parameter description.

## 20.22.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I[0], data input<br>I[n], n > 0, data input |
| **Output Link** | O, data output |

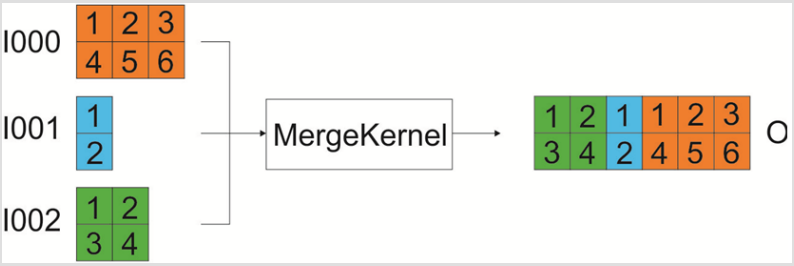
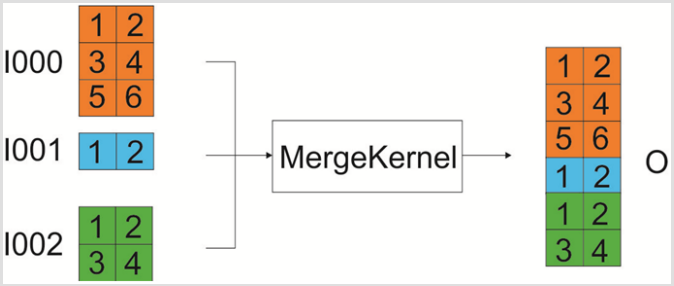
## 20.22.2. Supported Link Format

| Link Parameter | Input Link I[0] | Input Link I[n], n > 0 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | as I[0] | as I[0] |
| **Arithmetic** | {unsigned, signed} | as I[0] | as I[0] |
| **Parallelism** | any | as I[0] | as I[0] |
| **Kernel Columns** | any | horizontal: any<br>vertical: as I[0] | horizontal: sum of all kernel columns<br>vertical: as I[0] |
| **Kernel Rows** | any | horizontal: as I[0]<br>vertical: any | horizontal: as I[0]<br>vertical: sum of all kernel rows |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I[0] | as I[0] |
| **Color Format** | any | as I[0] | as I[0] |
| **Color Flavor** | any | as I[0] | as I[0] |
| **Max. Img Width** | any | as I[0] | as I[0] |
| **Max. Img Height** | any | as I[0] | as I[0] |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.22.3. Parameters

| Alignment | |
|---|---|
| **Type** | static write parameter |
| **Default** | horizontal |
| **Range** | [horizontal, vertical] |
| Parameter Alignment defines the ordering of the output vector. The vector is a concation of all inputs in either horizontal or vertical direction. | |

| **Alignment** |
| --- |
| Via parameter *Alignment*, you define if the number of rows or the number of columns has the same value for all kernels at the inputs.<br><br>**Horizontal:** If you select "horizontal" (default) the number of rows needs to be the same for the incoming kernels at all inputs. The number of columns you can define individually for each incoming kernel. In the output, the number of rows is the same as in the inputs. The number of columns will be the sum of the columns of all inputs.<br><br><br><br><br>**Vertical:** If you select "vertical" the number of rows can be defined individually for each incoming kernel. The number of columns needs to be the same for the incoming kernels at all inputs. In the output, the number of columns is the same as in the inputs. The number of rows will be the sum of the rows of all inputs.<br><br><br> |

## 20.22.4. Examples of Use

The use of operator MergeKernel is shown in the following examples:

- Section 12.8.1.2, 'Kirsch Filter'

  Examples - The Kirsch filter is a good edge detection filter for non directional edges.

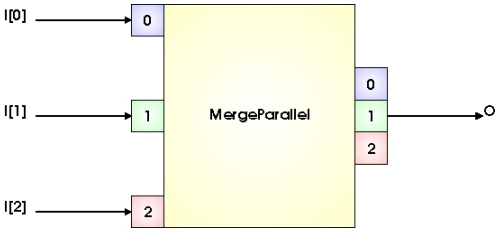
- Section 12.13.1, 'Histogram of Oriented Gradients (HOG)'

  Examples- Histogram of oriented Gradients

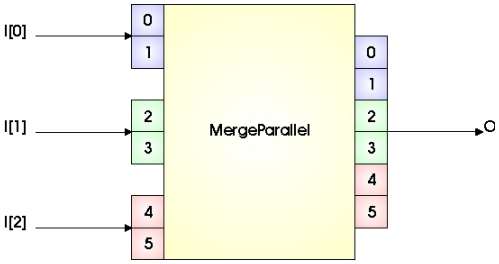
# 20.23. Operator MergeParallel

**Operator Library: Base**

The operator merges N input links to a single output link by concatenating the input links to parallelism components. The following images illustrate 2 cases: In case A, 3 input links of parallelism 1 are merged. In case B, 3 input links of parallelism 2 are merged.




The input parallel pixels are concatenated to a single output link of parallelism 3. The input link order determines the position of the corresponding pixel in the parallel output.




Note that input pixels in this case are also concatenated to the parallel output and are not interleaved. The 3 input links of parallelism 2 are concatenated to the output link of parallelism 6.

## 20.23.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input<br>I[n], n > 0, data input |
| **Output Link** | O, data output |

## 20.23.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I[n], n > 0 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | as I0 | as I0 |
| **Arithmetic** | {unsigned, signed} | as I0 | as I0 |
| **Parallelism** | any | as I0 | auto❷ |
| **Kernel Columns** | any | as I0 | as I0 |
| **Kernel Rows** | any | as I0 | as I0 |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | as I0 | as I0 |
| **Color Format** | any | as I0 | as I0 |
| **Color Flavor** | any | as I0 | as I0 |
| **Max. Img Width** | any | as I0 | auto❸ |

| Link Parameter | Input Link I0 | Input Link I[n], n > 0 | Output Link O |
|---|---|---|---|
| **Max. Img Height** | any | as I0 | as I0 |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The output parallelism is the sum of the input parallelism of all input links.
❸ The output maximum image width is the sum of the input maximum image widths of all input links.

## 20.23.3. Parameters

| MergeMode | |
|---|---|
| **Type** | static parameter |
| **Default** | Append |
| **Range** | {Append, Interleave} |

This parameter defines the way how pixel from the input links are arranged in the output link.

Append: All parallel pixel of each input link are appended.

Example: If you have 3 input links of parallelism 2 then you get an output link of parallelism 6. The first 2 pixel are taken from I0, then 2 pixels from I1, and finally 2 pixels from I2.

Interleave: Pixels are interleaved round robin over the input links.

Example: If you have 3 input links of parallelism 2 then you get an output link of parallelism 6. The parallel pixels in the output link get arranged as follows: pixel 0 of I0, pixel 0 of I1, pixel 0 of I2, pixel 1 of I0, pixel 1 of I1, and finally pixel 1 of I2.

## 20.23.4. Examples of Use

The use of operator MergeParallel is shown in the following examples:

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.3.2, 'Blob 2D'

  Examples - Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.9.6, 'Line Mirror'

Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

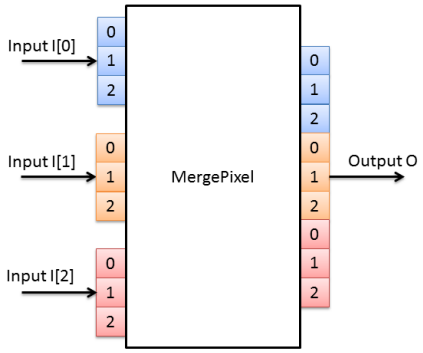
# 20.24. Operator MergePixel

**Operator Library: Base**

The operator MergePixel merges pixels of multiple input links of the same parallelism into one pixel of larger bit width. The bits of the inputs are concatenated. The bits of input I0 will be mapped to the lower bits at the output. All inputs may have varying bit widths. The output is always unsigned. If signed pixels are merged, the bits are simply reinterpreted and concatenated.

Each color component is merged separately.

The following figure shows the merge of three input links into one link.




## 20.24.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input<br>I[n], n > 0, data input |
| **Output Link** | O, data output |

## 20.24.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I[n], n > 0 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 63]❶ | [1, 63]❷ | auto❸ |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I0 | as I0 |
| **Kernel Columns** | 1 | as I0 | as I0 |
| **Kernel Rows** | 1 | as I0 | as I0 |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | as I0 | as I0 |
| **Color Format** | any | as I0 | as I0 |
| **Color Flavor** | any | as I0 | as I0 |
| **Max. Img Width** | any | as I0 | as I0 |

| Link Parameter | Input Link I0 | Input Link I[n], n > 0 | Output Link O |
|---|---|---|---|
| **Max. Img Height** | any | as I0 | as I0 |

❶❷ The range of the input bit width is [1, 63] for unsigned values. For signed inputs, the range is [2, 63]. The sum of the input bits of all inputs must be <= 64.
❸ The output bit width is the sum of the bit widths of all inputs. The output bit width must be <= 64.

## 20.24.3. Parameters

None

## 20.24.4. Examples of Use

The use of operator MergePixel is shown in the following examples:

- Section 10.1.1.4, 'RGB Camera Link Medium Area'

  Tutorial - Basic Acquisition

- Section 10.1.1.5, 'Grayscale Camera Link Full Area'

  Tutorial - Basic Acquisition

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.2.4, 'Simple Threshold Binarization'

  Simple thresholding for binarization.

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.3.2, 'Blob 2D'

  Examples - Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels.

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

- Section 12.8.1.1, 'Morphological Edge'

  Examples - A binary eroded image is compared with the original. An edge is detected if both differ.

# 20.25. Operator NOP

**Operator Library: Base**

The operator NOP (no operation) has no influence on the processed data in the VisualApplets design. Image data is not changed. The operator can be used to enhance the appearance and positioning of links and operators in a design. For example, it is can be used to label a link with a name.

Technically, the NOP operator outputs the registered input i.e. the operator is represented by a register stage. The use of these additional register stages in between a possible critical path might improve the timing of a design for failing period constraints or might enable the use of a higher clock frequency. The benefit of inserting a NOP operator into a design depends on the implementation of operators. For example, a long chain of O-type operators might cause a critical path, where a NOP operator will help to improve timing. As the operator is of type O, no balancing of delays is required for parallel paths by the user. Visual Applets will automatically perform this balancing if all synchronization rules have been followed. You cannot use the NOP operator to generate a delay.

## 20.25.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.25.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

## 20.25.3. Parameters

None

## 20.25.4. Examples of Use

The use of operator NOP is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

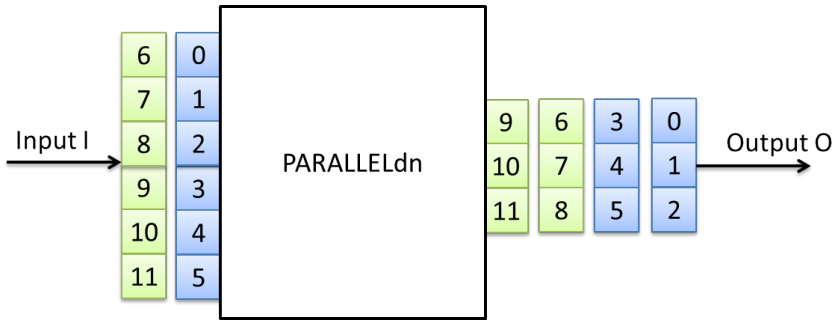
# 20.26. Operator PARALLELdn

**Operator Library: Base**

Library: Base

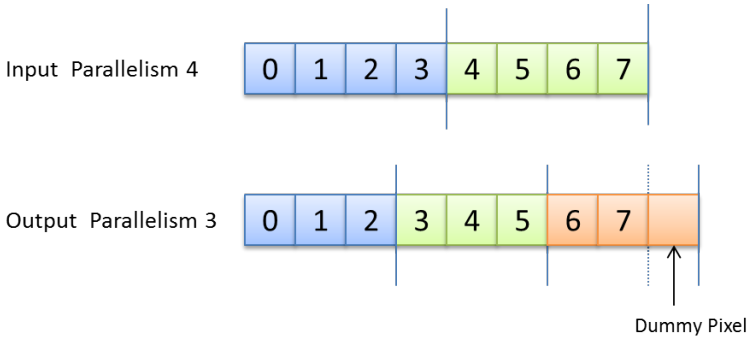
The operator PARALLELdn decreases the parallelism between the input link and the output link.

Following figure shows a parallel down conversion from parallelism 6 at the input down to parallelism 3 at the output.

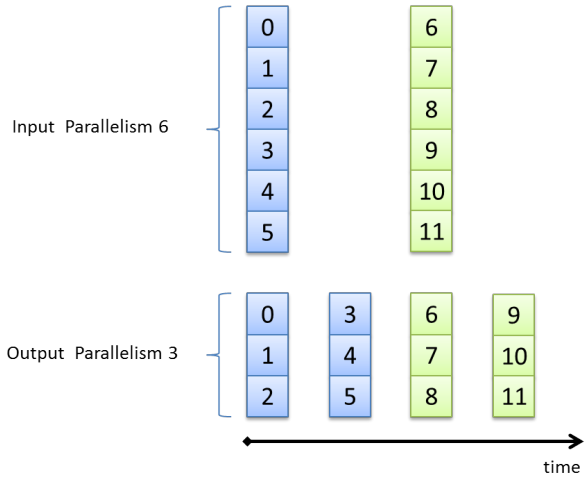



In VisualApplets, the width of an image line has to be a multiple of the parallelism. As the PARALLELdn operator allows any parallelism at its output it is possible that the width of a image line at the output is not a multiple of the output parallelism. In this case the operator will add dummy pixels to fill up the image line. The value of this dummy pixel is undefined. In VA simulation dummy pixels will be set to zero for better visibility.

The following figure shows in illustration of this dummy pixel insertion. As can be seen, the output parallelism of three cannot map the input line width of eight. Here, one extra dummy pixel is added so that the line width becomes nine.




Note that a parallelism decrease will result in a lower bandwidth of the output link compared to the input link. As the operator cannot buffer data, the reduced output bandwidth is also present at the input and might influence the precedent operators in the image pipeline. Technically, the input of the operator is closed/blocked for some clock cycles at a parallel down conversion. The following figure illustrates the timing of the operator.




The operator requires fewest resources if the input parallelism is an integer multiple of the output parallelism.

## 20.26.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.26.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | <= input parallelism |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | auto❷ |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

❷ The output image width must not exceed 2^31-1. The maximum image width at the output is automatically rounded to the next multiple of *O.Parallelism*. As a result, *I.MaxImgWidth* must not be greater than 2^31-1-O.Parallelism so that the rounded maximum image width at the output doesn't exceed 2^31-1.

## 20.26.3. Parameters

None

### 20.26.4. Examples of Use

The use of operator PARALLELdn is shown in the following examples:

- Section 3.6.7, 'Bandwidth Bottlenecks'

  Bandwidth Bottlenecks - Reducing the parallelism after the removal of pixels.

- Section 11.2, 'Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Penta Platform'

  Examples - Demonstration of how to use the operator

- Section 11.3, 'Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Quad Platform'

  Examples - Demonstration of how to use the operator

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

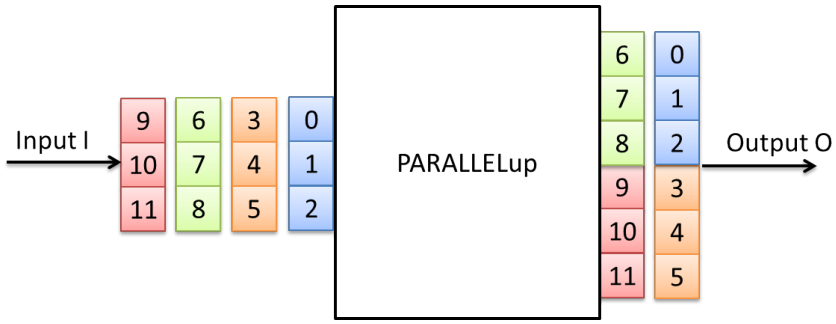
# 20.27. Operator PARALLELup

**Operator Library: Base**

Library: Base

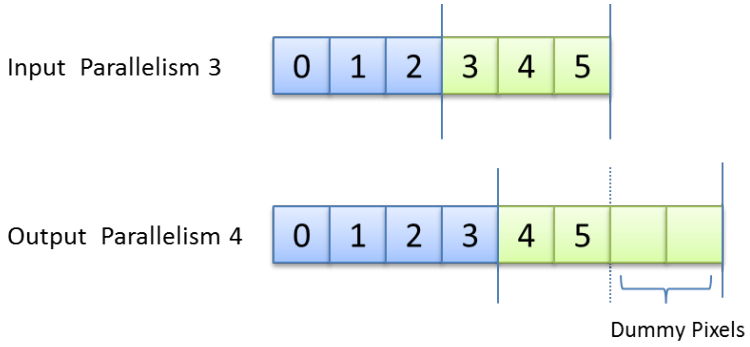
The operator PARALLELup increases the parallelism between the input link and the output link.

Following figure shows a parallel up conversion from parallelism 3 at the input to parallelism 6 at the output.




In VisualApplets, the width of an image line always has to be a multiple of the parallelism. As the PARALLELup operator allows any parallelism at its output it is possible that the width of an image line at the output is not a multiple of the output parallelism. In this case the operator will add dummy pixels to fill up the image line. The value of this dummy pixel is undefined. In VA simulation dummy pixels will be set to zero for better visibility.

The following figure shows in illustration of this dummy pixel insertion. As can be seen, the output parallelism of four cannot map the input line width of 6 pixels. Here, two extra dummy pixel are added so that the line width becomes eight.




The operator requires fewest resources if the output parallelism is an integer multiple of the input parallelism.

## 20.27.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.27.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | >= input parallelism |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | auto❷ |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

❷ The output image width must not exceed 2^31-1. The maximum image width at the output is automatically rounded to the next multiple of *O.Parallelism*. As a result, *I.MaxImgWidth* must not be greater than 2^31-1-O.Parallelism so that the rounded maximum image width at the output doesn't exceed 2^31-1.

## 20.27.3. Parameters

None

## 20.27.4. Examples of Use

The use of operator PARALLELup is shown in the following examples:

- Section 3.7.2.2, 'Parameter Editing'

  Design Parametrization - Invalid link property.

- Section 11.1, 'Functional Example for the *FrameBufferMultRoiDyn* Operator on the imaFlex CXP-12 Penta Platform'

  Examples - Demonstration of how to use the operator

- Section 11.2, 'Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Penta Platform'

  Examples - Demonstration of how to use the operator

- Section 11.3, 'Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Quad Platform'

  Examples - Demonstration of how to use the operator

- Section 12.4.2.2, 'Color Plane Separation Option 2 - Three Buffers, One DMA'

  Sequential output of the color planes using three image buffers and one DMA operator.

- Section 12.4.2.4, 'Color Plane Separation Option 4 - Sequential with Operator ImageBufferMultiRoI and a pre-sort of the Color Planes'

  Sequential DMA output of the color planes. The color separations is performed using operator ImageBufferMultiROI. An additional pre-sorting optimizes the bandwdith and resources.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

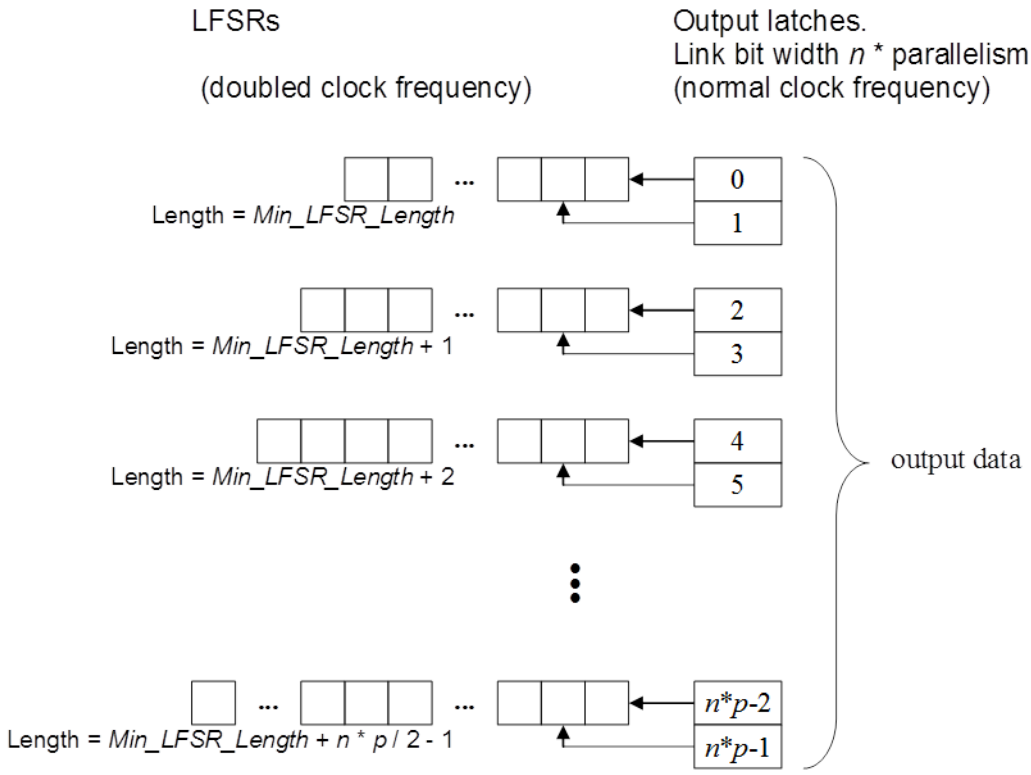
- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

# 20.28. Operator PseudoRandomNumberGen

**Operator Library: Base**

The operator PseudoRandomNumberGen generates a stream of N bit random numbers using linear feedback shift registers (LFSR). These LFSRs generate uniformly distributed binary random sequences. The length of the random sequence until it is repeated is defined by the number of registers of the feedback shift registers. A LFSR of 32 bit length will generate a sequence of 2^32 - 2 values. The implemented random number generator uses different LFSRs to generate the N bit random number at the operator output. This increases the sequence lengths and avoids correlations between the bits. The following figure illustrates the implementation of the pseudo random number generator.




To further improve the random number quality, the LFSRs are free running and not clock sourced by the pixel frequency. The timing of the results therefore is non-deterministic and hence, real random numbers are taken into the value generation which dramatically improves the quality.

The LFSRs are preinitialized with a seed on initialization of the applet. This seed is defined in VisualApplets using parameters *Seed0*, *Seed1* and *Seed2*. On operator instantiation, a software random number generator is used for the default seed values.

## 20.28.1. Usage

The operator supports different bit widths and parallelism. However, for high parallelism and bit width more resources are required. The output bit width can be adapted using the output link.

The operator input link is only used for synchronization. The data values on the input link are not used for data output generation.

Parameter *Min_LFSR_Length* is used to specify the length of the shortest LFSR. For each further LFSR another register stage is added. The maximum register length is 168.

To reduce the required resources of the operator, reduce parameter *Min_LFSR_Length*. If more than one operator is used with the same *Min_LFSR_Length* they will both generate the same sequence. If the seeds are different, the sequences are still the same, only the starting point of the sequence changed.

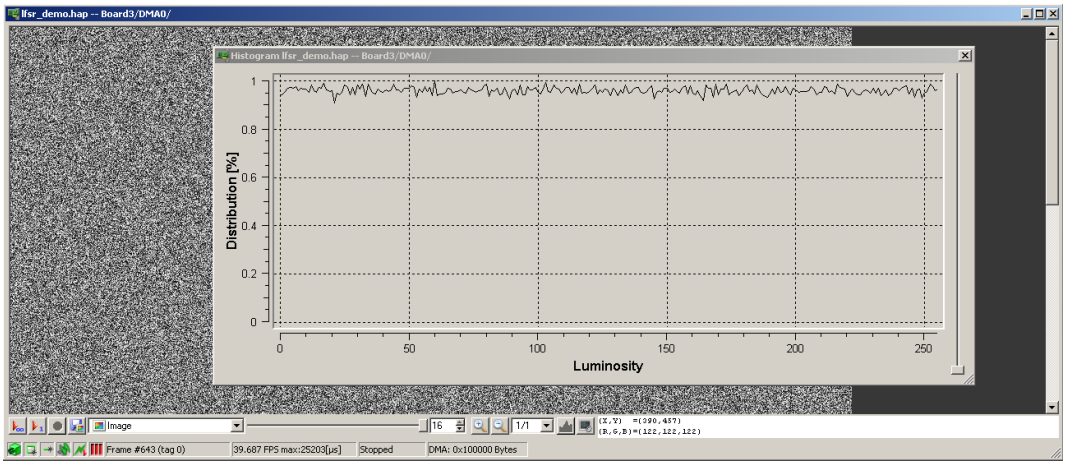
## 20.28.2. Quality of the Generator Random Numbers

A pseudo random number generator can never generate real random values. Due to the efficient generator implementation and the adding of non-deterministic timings, the quality of the generated values is very high. In the following, the results of two tests applied on the generator are presented to prove the quality of the implementation.

1. Test on Equidistribution:

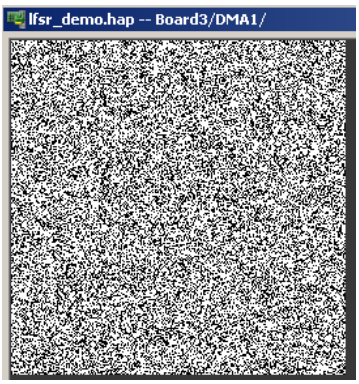
   A sequence of 8 bit random values is generated and the mean value is determined. The mean value should be around 127.5. From theory of the LFSR we know that sequences are always equidistributed (except value 0 in all registers).

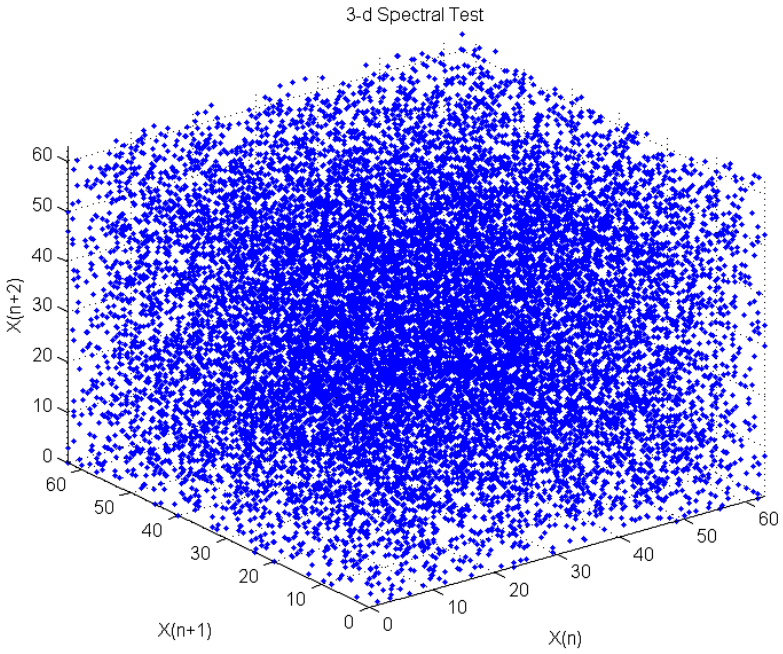
   If looking at a histogram of the generated values (for example in microDisplay), the uniform distribution can be seen.



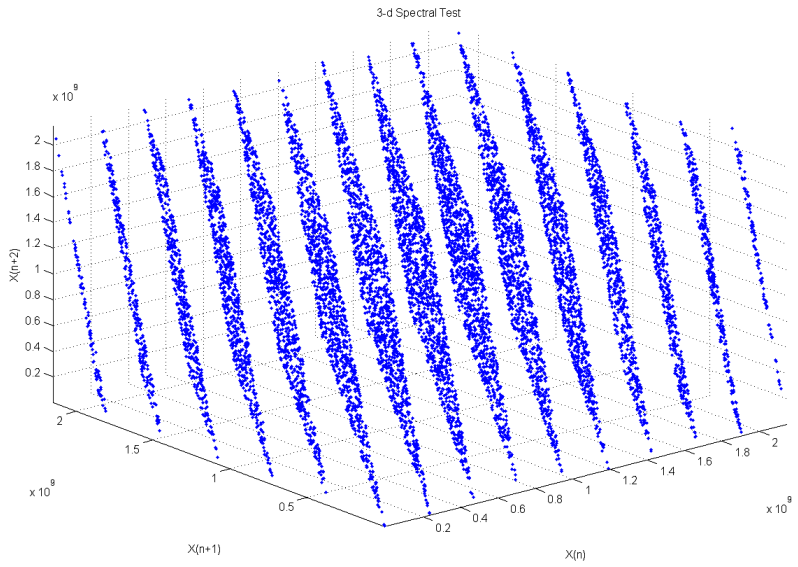

2. Spectral Test:

   A spectral test tests the generator on serial correlation. No patterns should be visible in the generated sequences. Bad generators show hyperplanes in the two- or three-dimensional space. The VisualApplets generator does not show any hyperplanes:






An example of a bad generator is shown in the next figure. It is the famous RANDU Generator which is mostly used for the rand() function in C programming languages.




## 20.28.3. VisualApplets Simulations with PseudoRandomNumberGen

The operator can fully be simulated in VisualApplets. However, the non-deterministic hardware implementation cannot be implemented in software for simulation. Therefore, a very simple random number generator with bad quality is used in the simulation. (rand() function of the MS VisualStudio) If seeds are not changed, the operator will always generate the same sequences.

## 20.28.4. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |

| Property | Value |
|---|---|
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.28.5. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | any❶ | [1, 64]❷ |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I❸ |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned gray values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷❸ $$Min_LFSR_Length + \frac{OutputBitWidth \times OutputParallelism}{2} - 1 \leq 168$$

## 20.28.6. Parameters

| Min_LFSR_Length | |
|---|---|
| **Type** | static parameter |
| **Default** | random |
| **Range** | [3, 168] |

This parameter specifies the length of the shortest LFSR. Higher values cause a higher resource consumption.

Note the follwing constraint:

$$Min_LFSR_Length + \frac{OutputBitWidth \times OutputParallelism}{2} - 1 \leq 168$$

| Seed0 | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | random |
| **Range** | [0, 2^64 - 1] |

As explained in the operator introduction, the seeds are used to pre-initialize the LFSRs. On operator instantiation these values are initialized using a software generated random number. The values can be chnaged during acquisition. The values of all three parameters are latched when parameter *Seed2* is modified.

| Seed1 | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | random |
| **Range** | [0, 2^64 - 1] |

| **Seed1** |
| --- |
| As explained in the operator introduction, the seeds are used to pre-initialize the LFSRs. On operator instantiation these values are initialized using a software generated random number. The values can be chnaged during acquisition. The values of all three parameters are latched when parameter *Seed2* is modified. |

| **Seed2** | |
| --- | --- |
| **Type** | static/dynamic read/write parameter |
| **Default** | random |
| **Range** | [0, 2^64 - 1] |
| As explained in the operator introduction, the seeds are used to pre-initialize the LFSRs. On operator instantiation these values are initialized using a software generated random number. The values can be chnaged during acquisition. The values of all three parameters are latched when parameter *Seed2* is modified. | |

## 20.28.7. Examples of Use

The use of operator PseudoRandomNumberGen is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

  Examples - Demonstration of how to use the operator

# 20.29. Operator SampleDn

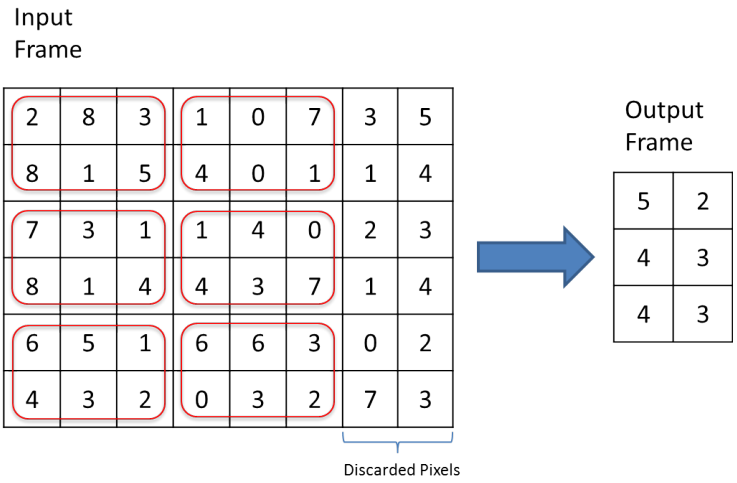
**Operator Library: Base**

The operator SampleDn reduces the image size by downsampling the input image. The image size downsampling factor can be individually defined for the width and height using any positive integer number up to 128. This downsmpling factor is defined using parameters *XSampleDown* and *YSampleDown*. The output pixels are computed by averaging the joint pixels.

The operator functions properly only on rectangular images, i.e. images whose lines are of equal length. Images with variable line length within the same image will lead to wrong results.

If the down sampling parameters are set to a value that exceeds the width/height of the input image, the resulting image will be an empty image.

If the actual image width/height is not divisible by the correspondent downsampling parameter value, the operator will discard the remaining pixels. See the following figure for explanation.




If the width of an incoming image divided by *XSampleDown* is not a multiple of the parallelism and is smaller than the calculated max. image width of the output then the image lines are expanded by appending dummy pixels so the resulting line length is a multiple of the parallelism. The value of the inserted dummy pixels is undefined.

**Operator Restrictions**

- The lines of each input image must have the same length. Thus images with varying line lengths are not allowed.
- For color pixels, each color component is processed individually.
- If the actual image width is less than *XSampleDown*, an empty image is output.
- If the actual image height is less than *YSampleDown*, an empty image is output.

## 20.29.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.29.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | auto❷ |
| **Max. Img Height** | any | auto❸ |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

❷❸ The maximum image dimension at the output is automatically determined according to the following formulas:

$$OutputMaxImgWidth = \left\lfloor \frac{InputMaxImageWidth}{XSampleDown \times Parallelism} \right\rfloor \times Parallelism$$

$$OutputMaxImgHeight = \left\lfloor \frac{InputMaxImageHeight}{YSampleDown} \right\rfloor$$

## 20.29.3. Parameters

| **XSampleDown** | |
|---|---|
| **Type** | static parameter |
| **Default** | 1 |
| **Range** | [1, 128] |
| This parameter defines the down sampling factor for the image width.<br>The downsampling is limited to<br>$2^{64} - 1 \geq \left(2^{BitWidth} - 1\right) \times XSampleDown \times YSampleDown$ | |

| **YSampleDown** | |
|---|---|
| **Type** | static parameter |
| **Default** | 1 |
| **Range** | [1, 128] |
| This parameter defines the down sampling factor for the image height.<br>The downsampling is limited to<br>$2^{64} - 1 \geq \left(2^{BitWidth} - 1\right) \times XSampleDown \times YSampleDown$ | |

## 20.29.4. Examples of Use

The use of operator SampleDn is shown in the following examples:

- Section 12.9.1, 'Downsampling'

Examples - The input image is downsampled i.e. reduced in size by 4x4.

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

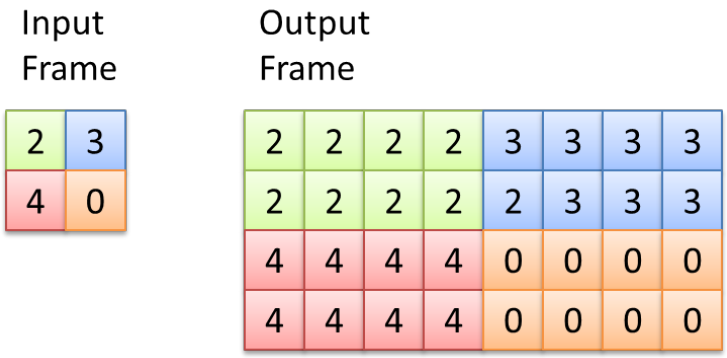
# 20.30. Operator SampleUp

**Operator Library: Base**

Library: Base

The operator SampleUp increases the image size by upsampling the input image. The image size upsampling factor can be individually defined for the width and height in steps 1, 2, 4, 8 and 16. The upsampling factor is defined using parameters *Expand_X* and *Expand_Y*. The output pixels are computed by duplicating pixels (nearest neighbor method).

See the following figure for explanation of the operator's output. In this example, an upsampling by 4 in x-direction and 2 in y-direction is performed.




Note that the parallelism at the output is not increase even though more pixels have to be transported on the link. Thus this operator cannot process the input image data at its parallelism.

**Operator Restrictions**

- The lines of each input image must have the same length. Thus images with varying line lengths are not allowed.
- Empty images or images with empty lines are not allowed.

## 20.30.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.30.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | {1, 2, 4, 8, 16, 32, 64, 128} | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | auto❷ |
| **Max. Img Height** | any | auto❸ |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].
❷ The output image width must not exceed 2^31 - 1.
❸ The output image height must not exceed 2^31 - 1.

## 20.30.3. Parameters

| Expand_X | |
|---|---|
| **Type** | static parameter |
| **Default** | UpSampleXBy1 |
| **Range** | {UpSampleXBy1, UpSampleXBy2, UpSampleXBy4, UpSampleXBy8, UpSampleXBy16} |
| This parameter defines the upsampling factor for the image width. | |

| Expand_Y | |
|---|---|
| **Type** | static parameter |
| **Default** | UpSampleYBy1 |
| **Range** | {UpSampleYBy1, UpSampleYBy2, UpSampleYBy4, UpSampleYBy8, UpSampleYBy16} |
| This parameter defines the upsampling factor for the image height. | |

## 20.30.4. Examples of Use

The use of operator SampleUp is shown in the following examples:

- Section 12.13.1, 'Histogram of Oriented Gradients (HOG)'

  Examples- Histogram of oriented Gradients

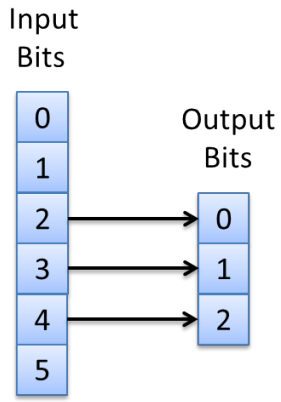
# 20.31. Operator SelectBitField

**Operator Library: Base**

The module SelectBitField makes it feasible to select a field of succeeding bits form the input link. Define the required output bit width using the output link. Additionally, you can define the bit offset using parameter *BitOffset*. If the input arithmetic is signed, the values will be reinterpreted and always output as unsigned values.

Each color component is sliced separately.

See the following figure for explanation. In this example, the output transports 3 bits cut from the 6 bit input values with an offset of two. For example, input value 20 will become 5.




## 20.31.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.31.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | [1, 64]❶ |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | {1, 2, 4, 8, 16, 32, 64, 128} | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The output bit width has to be in

$$OutputBitWidth \leq InputBitWidth - BitOffset$$

### 20.31.3. Parameters

| BitOffset | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, InputBitWidth - OutputBitWidth] if gray [0, (InputBitWidth - OutputBitWidth) / 3] if color |
| This parameter defines the bit offset for the first bit to be used. This first bit is mapped to bit position 0 at the output link. For color formats, the offset is handled per component. | |

### 20.31.4. Examples of Use

The use of operator SelectBitField is shown in the following examples:

- Section 10.1.1.4, 'RGB Camera Link Medium Area'

  Tutorial - Basic Acquisition

- Section 10.1.1.5, 'Grayscale Camera Link Full Area'

  Tutorial - Basic Acquisition

- Section 11.8, 'Example for the *DMAFromPC* Operator on the imaFlex CXP-12 Quad Platform'

  Examples - Demonstration of how to use the operator using the example of shading correction

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

# 20.32. Operator SelectComponent

**Operator Library: Base**

The operator SelectComponent enables the extraction of a single color component from the input link and makes this value available at the output link as a gray pixel. Use parameter *Component* to select the required component.

## 20.32.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.32.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [3, 63] unsigned, [6, 63] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | VAF_GRAY |
| **Color Flavor** | {FL_RGB, FL_HSI, FL_YUV, FL_LAB, FL_XYZ} | FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The output bit width is

$$OutputBitWidth = InputBitWidth/3$$

## 20.32.3. Parameters

| **Component** | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 2] |
| This parameter is used to select one of the color component from the input, e.g if the input flavor is RGB Component=0 selects red, 1 select green, and 2 selects blue. | |

## 20.32.4. Examples of Use

The use of operator SelectComponent is shown in the following examples:

- Section 12.13.4, 'Normalized Cross Correlation'

  Examples-

# 20.33. Operator SelectFromParallel

**Operator Library: Base**

The operator SelectFromParallel extracts a single parallelism component from the input link and makes this value available at the output link. Use parameter *ParNum* to select the required component. The parallelism at the output link is always one. Only the selected parallelism components will be output.

For example, if an input parallelism of four is used and parallelism component 1 is selected, the operator will output pixels with index 1, 5, 9, 13, ...

## 20.33.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.33.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | 1 |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | auto❷ |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

❷ The maximum output image width is:

$$OutputMaxImgWidth = InputMaxImgWidth / InputParallelism$$

## 20.33.3. Parameters

| ParNum | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, InputParallelism - 1] |
| This parameter defines which parallel component of the parallel transmitted pixels of the input link is forwarded to the output. | |

## 20.33.4. Examples of Use

The use of operator SelectFromParallel is shown in the following examples:

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

# 20.34. Operator SelectROI

**Operator Library: Base**

The operator SelectROI extracts a rectangular region of interest (ROI) from the frames at the input link. The ROI is specified by using 4 parameters X_Offset, X_Length, Y_Offset and Y_Length. X parameters are to be specified in pixels. Y parameters are to be specified in lines.

If it is required to set the ROI coordinates using input links instead of parameters, operator *DynamicROI* must be used.

The operator will finalize the output image as soon as the ROI is completed. If the ROI is smaller than the input image this can result in a behavior that the output image is finalized earlier to the finalization of the input image. This is in contrast to operators *RemoveLine* or *RemovePixel* which have to wait for the last line / pixel.

**Operator Restrictions**

The sum of X_Offset and X_Length parameters must not exceed the maximal link image width. The sum of Y_Offset and Y_Length parameters must not exceed the maximal link image height.

Changing the dynamic ROI parameters is allowed when the process i.e. the acquisition is stopped only. Changes while the acquisition is running will have no effect on the ROI. This avoids invalid ROIs when the parameters are changed.

The lines of each input image must have the same length. Thus images with varying line lengths are not allowed.

- In VisualApplets it is required that the number of pixels in a line i.e. the line width is always a multiple of the parallelism. When the X_Length is not divisible by the link parallelism the operator will insert dummy pixels to fill up the last parallel word, e.g. if the link parallelism is 2, X_Offset is 0 and X_Length is 5, the operator will output 6 pixel where the last pixel is a dummy pixel. The value of that dummy pixel is undefined. In VA simulation dummy pixels will be set to zero for better visibility.
- When the input frame is smaller than the specified ROI the operator will output only the available part of the ROI.
- When using the operator in 1D mode, the Y parameters have no effect on the operator. The operator will cut off ROIs only in X direction.

## 20.34.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.34.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.34.3. Parameters

| X_Offset | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, MaxImgWidth - X_Length] |
| This parameter value (together with X_Offset) defines the position of the left edge of the ROI in pixels. | |

| Y_Offset | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, MaxImgHeight - Y_Length] |
| This parameter value (together with Y_Offset) defines the position of the upper edge of the ROI in pixels. | |

| X_Length | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 1024 |
| **Range** | [1, MaxImgWidth - X_Offset] |
| This parameter defines the width of the ROI. | |

| Y_Length | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 1024 |
| **Range** | [1, MaxImgHeight - Y_Offset] |
| This parameter defines the height of the ROI. | |

## 20.34.4. Examples of Use

The use of operator SelectROI is shown in the following examples:

- Section 12.9.5, 'Moments in Image Processing'

  Example - Calculates image moments orientation and eccentricity

# 20.35. Operator SelectSubKernel

**Operator Library: Base**

The operator SelectSubKernel extracts a rectangular subset of the kernel matrix at the input link. Similar to the ROI an offset for the first element of the new kernel can be defined using parameters *FirstROW* and *FirstCOL*. The size (width/height) of the new kernel can be chosen at the output link by setting the kernel size of the link.

Often, this operator is used to extract the central element of a kernel which mostly is the original pixel value.

## 20.35.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.35.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | any❷ |
| **Kernel Rows** | any | any❸ |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷❸ Range is defined as follows:

$$OutputKernelRows \leq InputKernelRows - FirstRow$$

$$OutputKernelColumns \leq InputKernelColumns - FirstColumns$$

## 20.35.3. Parameters

| FirstROW | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, InputKernelRows - OutputKernelRows] |
| This parameter defines the first row for the new kernel. | |

| FirstCOL | |
|---|---|
| **Type** | static parameter |

| FirstCOL | |
|---|---|
| **Default** | 0 |
| **Range** | [0, InputKernelColumns - OutputKernelColumns] |
| This parameter defines the first column for the new kernel. | |

## 20.35.4. Examples of Use

The use of operator SelectSubKernel is shown in the following examples:

- Section 3.6.6, 'Timing Synchronization'

  Synchronization - Avoiding deadlocks.

- Section 12.1.6, 'Laser Pointer Detection'

  Examples - A convolution with high intensity spot coefficients is made. For results above threshold, the respective pixels are dyed in red.

- Section 12.4.1.1, 'Nearest Neighbor De-Mosaicing'

  Examples - Nearest Neighbor Bayer De-Mosaicing

- Section 12.4.1.6, 'Edge Sensitive Bayer De-Mosaicing Algorithm'

  Examples - Edge Sensitive Laplace Bayer De-Mosaicing filter

- Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche'

  Examples - Laroche Bayer De-Mosaicing filter

- Section 12.4.1.8, 'Modified Laroche Bayer De-Mosaicing Algorithm '

  Examples - Ressource Optimized Laroche Bayer De-Mosaicing filter

- Section 12.8.1.1, 'Morphological Edge'

  Examples - A binary eroded image is compared with the original. An edge is detected if both differ.

- Section 12.8.4.3, 'Filter Sub Kernels'

  Examples - Shows how to extract a sub kernel from a filter to obtain the original image data. This example performs a simple local adaptive binarization.

- Section 12.8.5.1, 'High Boost Sharpening Filter'

  Examples - A high boost Laplace filter for sharpening

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

# 20.36. Operator SetDimension

**Operator Library: Base**

In VisualApplets designs, images of size less equal than the maximum image dimensions of a link can be processed on these links. The maximum image dimension on a link is set using the link properties *Max Image Width* and *Max Image Height*. Images transported on a link must not exceed this image dimensions. If the image size exceeds the maximum allowed image dimensions, operators will not work correct.

The maximum image dimension link properties should always be set to the minimum required values i.e. the maximum expected image dimensions of the link. For example, if an operator selects a ROI from an image. Due to the dynamic ROI parameters, the resulting output images will not have a fixed image size. If the user knows that the ROI size will not exceed a specific value, the link properties can manually be changed. This can be done by operator SetDimension.

Operator SetDimension overwrites the link properties maximum image width and maximum image height. The operator only changes this logic link property. There is no influence on the transported data in hardware. Always use this operator with care and ensure that the image size limitations are not exceeded.

## 20.36.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 20.36.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.36.3. Parameters

None

## 20.36.4. Examples of Use

The use of operator SetDimension is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

# 20.37. Operator SplitComponents

**Operator Library: Base**

The operator SplitComponents separates the components of a color stream at the input link into three separate gray image streams at the output. The opposite of this operator is Section 20.21, 'MergeComponents'.

## 20.37.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Links** | O0 (dynamic), data output color component {R, H, Y, L, X}<br>O1 (dynamic), data output color component {G, S, U, A, Y}<br>O2 (dynamic), data output color component {B, I, V, B, Z} |

## 20.37.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O0 (dynamic) |
|---|---|---|
| **Bit Width** | [3, 63] unsigned, [6, 63] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | VAF_GRAY |
| **Color Flavor** | {FL_RGB, FL_HSI, FL_YUV, FL_LAB, FL_XYZ | FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O1 (dynamic) | Output Link O2 (dynamic) |
|---|---|---|
| **Bit Width** | auto❷ | auto❸ |
| **Arithmetic** | as I | as I |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

❶❷❸The output bit width is the input bit width divided by three.

## 20.37.3. Parameters

None

## 20.37.4. Examples of Use

The use of operator SplitComponents is shown in the following examples:

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend'

  Tutorial - Split and Merge Color components.

- Section 10.1.1.4, 'RGB Camera Link Medium Area'

  Tutorial - Basic Acquisition

- Section 11.6, 'Functional Example for the *JPEG_Encoder_Color_iF* User Library Element on the imaFlex CXP-12 Quad Platform'

  Examples - Demonstration of how to use the operator

- Section 11.7, 'Functional Example for the *JPEG_Encoder_Color_iF_Penta* User Library Element on the imaFlex CXP-12 Penta Platform'

  Examples - Demonstration of how to use the operator

- Section 12.4.1.4, 'Bayer 3x3 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.2.1, 'Color Plane Separation Option 1 - Three DMAs'

  Splitting the RGB color planes into three DMA channel outputs.

- Section 12.4.2.2, 'Color Plane Separation Option 2 - Three Buffers, One DMA'

  Sequential output of the color planes using three image buffers and one DMA operator.

- Section 12.4.2.3, 'Color Plane Separation Option 3 - Sequential with Operator ImageBufferMultiRoI'

  Sequential DMA output of the color planes. The color separations is performed using operator ImageBufferMultiROI.

- Section 12.4.2.4, 'Color Plane Separation Option 4 - Sequential with Operator ImageBufferMultiRoI and a pre-sort of the Color Planes'

  Sequential DMA output of the color planes. The color separations is performed using operator ImageBufferMultiROI. An additional pre-sorting optimizes the bandwdith and resources.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.4.3, 'HSL Color Classification'

  Examples - Color Classification is very simple on HSL images. The applet converts the RGB image into an HSL image and performs a color classification. The hue is filtered using a lookup table. Moreover, the saturation and lightness is thresholded using custom threshold values.

- Section 12.6.4, 'Manual Image Injection'

  Example - For debugging purposes images can be inserted manually.
- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

# 20.38. Operator SplitKernel

**Operator Library: Base**

The operator splits the N x M kernel components of the input link into j = N * M output links. The amount of output links j has to be specified on operator initialization. The number of output links has to be equal to the number of kernel components. If the input kernel size changes, a new instantiation of the operator is required. The opposite of this operator is Section 20.22, 'MergeKernel'.

## 20.38.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O[0, j-1], data output |

## 20.38.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O[0, j-1] |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.38.3. Parameters

None

## 20.38.4. Examples of Use

The use of operator SplitKernel is shown in the following examples:

- Section 12.9.7, 'Shear of an Image'

  Example - Line Shear example with linear interpolation.

- Section 12.13.1, 'Histogram of Oriented Gradients (HOG)'

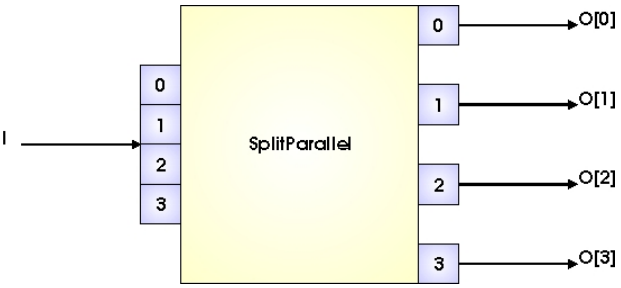
  Examples- Histogram of oriented Gradients

# 20.39. Operator SplitParallel

**Operator Library: Base**

The operator splits the input link of parallelism degree N into N output links of parallelism degree 1. The amount of output links N has to be specified on operator instantiation. The number of links has to be equal to the number of parallel pixels of the input link and cannot be changed after operator instantiation. If the input parallelism changes, a new instantiation of the operator is required. The opposite of this operator is Section 20.23, 'MergeParallel'.

The following figure illustrates an example for input parallelism four.




## 20.39.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O[0, N-1], data output |

## 20.39.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O[0, N-1] |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | 1 |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 20.39.3. Parameters

None

### 20.39.4. Examples of Use

The use of operator SplitParallel is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.9.6, 'Line Mirror'

  Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

- Section 12.9.9, 'Tap Geometry Sorting'

  Examples - Scaling A Line Scan Image

# 20.40. Operator Trash

**Operator Library: Base**

The operator Trash is a data sink for unused links.

If modules generate output links that are not required in a particular design or if some paths are not implemented yet, it is necessary to terminate open output links. Connecting open outputs to the input of a Trash module terminates these outputs.

## 20.40.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |

## 20.40.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | [1, 64]❶ |
| **Arithmetic** | {unsigned, signed} |
| **Parallelism** | any |
| **Kernel Columns** | any |
| **Kernel Rows** | any |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} |
| **Color Format** | any |
| **Color Flavor** | any |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 6] and for signed color, the range is [6, 63].

## 20.40.3. Parameters

None

## 20.40.4. Examples of Use

The use of operator Trash is shown in the following examples:

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

# 21. Library Blob




The *Blob* library includes operators for segmentation, object detection and feature extraction.



**Availability**

To use the **Blob** library, you need either a **Segmentation and Classification Library** license, or the **VisualApplets 4** license.

Object detection is of central importance for image analysis and classification. The object detection possibilities in VisualApplets allow you to design a huge range of image processing applications that run at highest speed grades in FPGA hardware. Using the *Blob Analysis* operators, you can realize complete image processing applications - including pre-processing, image analysis, and classification - in hardware. This offers you great options for solving computationally intensive high-speed applications. Blob analysis is a fundamental method for detecting objects using connected components in binary images. Blob analysis is strongly used in machine vision applications. The following sections introduce blob analysis and explain the features of the VisualApplets implementation.

## 21.1. Definition

A Blob Analysis operation detects objects in binary images and describes these objects using geometrical and statistical features. Figure 21.1, 'Objects Visualized by Colored Boxes' shows a binary image consisting of four objects. (In the figure, these four objects are visualized by colored boxes.)

In blob analysis, an object is defined by a set of pixels which differ from the background and are in direct neighborhood. Therefore, at first all pixels of an image need to be transformed into a binary image. Here, Visual Applets allows the design of various methods from simple thresholding up to complex adaptive binarization algorithms. Next, all black pixels of the resulting binary image are assumed to be background pixels, whereas all white pixels are assumed to be foreground pixels and thus to belong to an object.

Every white (foreground) pixel is allocated to an object. An object is defined to be a set of neighbored foreground pixels. Thus, if a white (foreground) pixel is in direct neighborhood to another white pixel, they belong to the same object. Because of this data interpretation, the image shown in Figure 21.1, 'Objects Visualized by Colored Boxes' results in four objects.

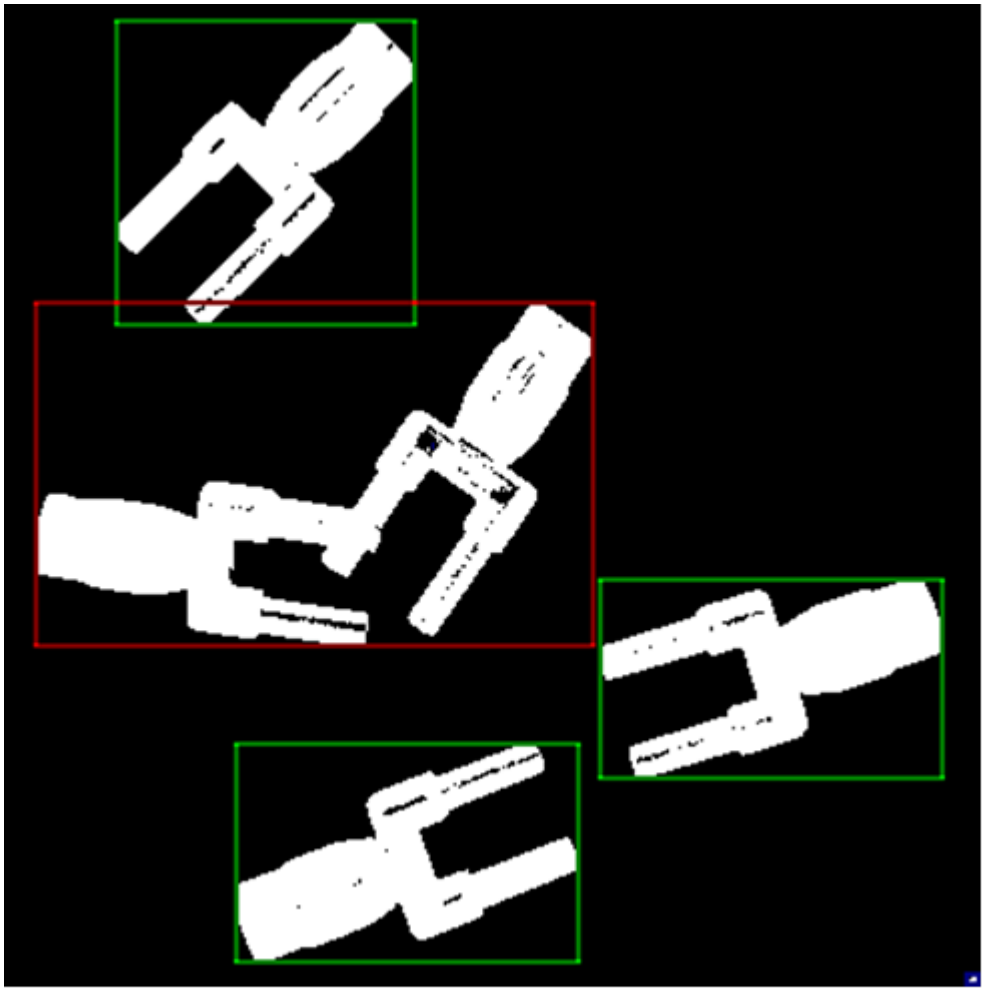


Figure 21.1. Objects Visualized by Colored Boxes

If a pixel is interpreted as belonging to a specific object depends on the relation of the pixel to its neighbor pixels.

Each pixel I(u,v) of the binary image has four pixels in its neighborhood which are directly connected. This relation is called **4-connected neighborhood**. If a foreground pixel I(u,v) has one or more other foreground pixels in its 4-connected neighborhood, the pixels belong to the same object.

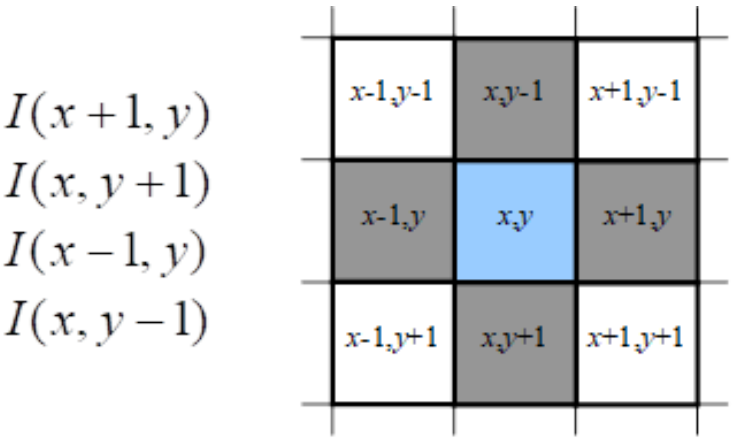



Figure 21.2. 4-Connected Neighborhood

Besides the direct neighborhood, a pixel can be connected to another pixel **diagonally**. This relation is called an 8-connected neighborhood:

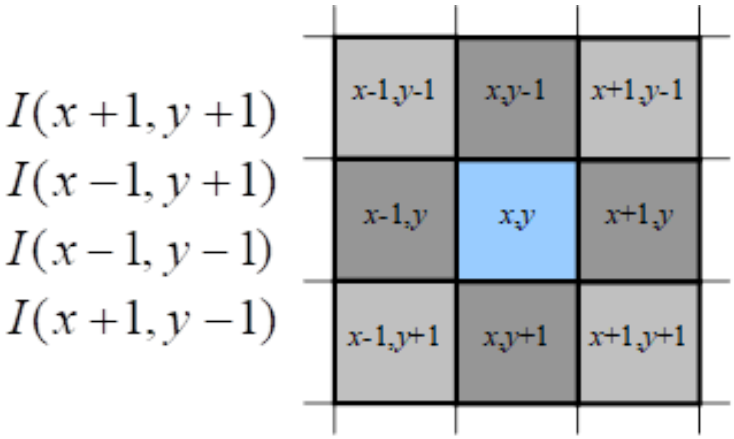



Figure 21.3. 8-Connected Neighborhood

The blob analysis operators can be parameterized to work with either 4-connected or 8-connected neighborhoods.

In general, an 8-connected neighborhood results in better object detection as even weakly connected objects are connected and edges are smoothened.

Example:

The image below shows a cutout of an image. On the left-hand side, the result of a blob analysis using a 4-connected neighborhood is displayed. On the right-hand side, the result of a blob analysis using an 8-connected neighborhood is displayed. (Object allocation is visualized via color.) The neighborhood parametrization (4-connected versus 8-connected) results in different allocations of pixels to objects: 4-connected neighborhood applied on the image results in six objects, whereas 8-connected neighborhood results in four objects only.

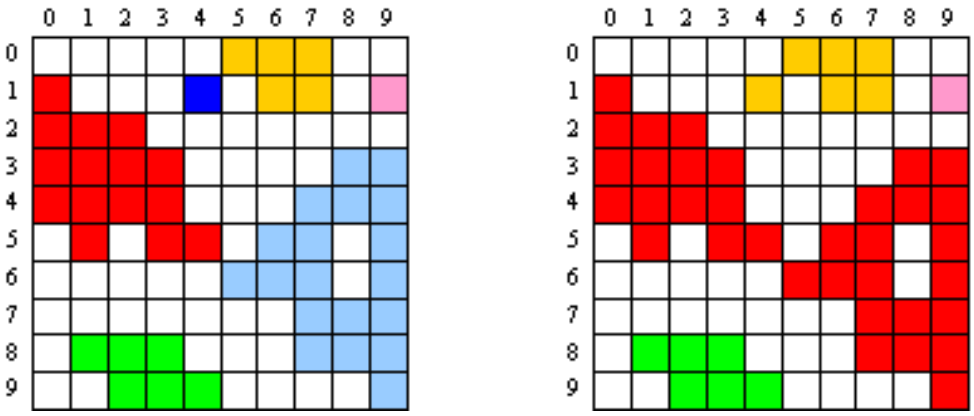



Figure 21.4. Pixels allocated to objects in a 4-connected neighborhood (left) and an 8-connected neighborhood (right). All colored pixels represent foreground pixels where their allocation to objects is visualized by differing colors.

## 21.2. Definition of Object Features

A blob analysis extracts features of the detected objects. Images of the objects themselves are not provided. The blob analysis operators offer the calculation of the most common object features which describe the objects using geometrical and statistical methods. Respectively, the features

- area
- bounding box
- center of gravity
- contour length

are supported. A comprehensive definition of each feature will be presented in the following.

### 21.2.1. Area

Each object of an image can be described by a list of the coordinates of all foreground pixels included in the object $p_i = (x_i, y_i)$. Therefore, the object or its region R can be described by

$$R = \{p_1, p_2, ..., p_n\} = \{(x_1, y_1), (x_2, y_2), ...(x_n, y_n)\}$$

The area of an object is the number of pixels of the object, i.e., the size of the region R.

$$Area(R) = A = N$$

Or in other words: The area of an object is the sum of all its pixels. This feature is often used to distinguish between objects of varying sizes.

### 21.2.2. Bounding Box

The bounding box of a region R is the minimum paraxial rectangle which fits over the object. It is a simple but very useful feature to describe the position of an object in the image.

$$B = BoundingBox(R) = (x_{min}, y_{min}, x_{max}, y_{max})$$

This feature provides four values for each object, described in VisualApplets as

$$X0 = x_{min}$$
$$X1 = x_{max}$$
$$Y0 = y_{min}$$
$$Y1 = y_{max}$$

This feature is different for the Blob Analysis 1D operator as no absolute coordinates for objects can be given. Please see Section 21.4, 'Blob_Analysis_1D' for explanation.

### 21.2.3. Center of Gravity

The center of gravity is calculated in x- and y-direction and is the sum of all absolute coordinates of an object. The point of origin for each object is the top left corner of an image.

$$C_x^* = \sum_{x \in R} x$$

and

$$C_y^* = \sum_{y \in R} y$$

Please note that the center-of-gravity values are not normalized. They need to be divided by object area after the blob analysis. To perform this division, use the DIV operator found in the *arithmetics* library. Alternatively, the division can be performed on the host PC.

$$C_x = \frac{C_x^*}{A}$$

and

$$C_y = \frac{C_y^*}{A}$$

This feature is different for the Blob Analysis 1D operator. Please see Section 21.4, 'Blob_Analysis_1D' for explanation.

### 21.2.4. Contour Length

This feature is defined by the length of all contours of an object. Included are outer and inner edges (the latter caused by holes).

*Contour Length* differentiates between a 4-connected neighborhood and an 8-connected neighborhood. If the blob analysis operator is set to detect objects using a 4-conneted neighborhood, the contour length is determined by scanning the edges of an object in orthogonal directions only. If the blob analysis operator is set to 8-connected neighborhood, the contour length also considers diagonally connected object edges. Thus, depending on the neighborhood setting of the operator, the feature provides two different results.

The two figures below illustrate the difference. Both figures show an image containing a single object.

In the upper figure, the object is detected using a 4-connected neighborhood. For this object, the contour length sums up all pixels located at object edges. Only horizontal and vertical pixel connections are considered. Therefore, the object results in a contour length of 30x orthogonal length.

In the lower figure, the blob analysis uses an 8-connected neighborhood. The contour length also considers diagonal edges. The object results in an contour length of 14x orthogonal length + 8x diagonal lengh.

The results differ in contour length.

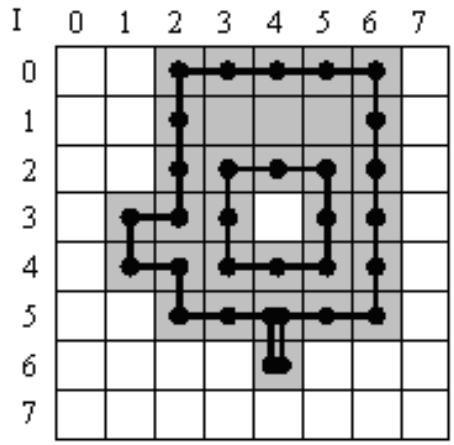



Figure 21.5. 4-connected neighborhood: Contour Orthogonal = 30, Diagonal = 0

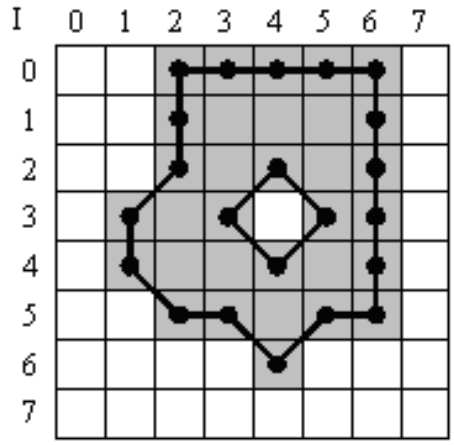



Figure 21.6. 8-connected neighborhood: Contour Orthogonal = 14, Diagonal = 8

The edge connecting two neighboring pixels diagonally has the contour length $c_i = \sqrt{1^2 + 1^2} = \sqrt{2}$ which represents the hypotenuse of an isosceles triangle. This square root operation is not performed by the blob analysis operators. Make sure it is performed afterwards in your design. The total contour length can be obtained by

$$C_{total} = C_{orthogonal} + \sqrt{2} * C_{diagonal}$$

The diagonal component of the contour length is very important for objects with edges that run not in parallel to the image edges.

For example, a circle having a radius of 5.5 pixels will have a perimeter of $P = 2 * \pi * r = 34.56$ . The figures below show this circle. In the left-hand figure, the perimeter is calculated using an 8-connected neighborhood. In the right-hand figure, the perimeter is calculated using an 4-connected neighborhood. The example shows that the result of the calculation using the 8-connected neighborhood (32.97) is close to the theoretic result (34.56), The result of the calculation using the 4-connected neighborhood (40) is rather poor - it is equal to the perimeter of a rectangle having the same diameter.

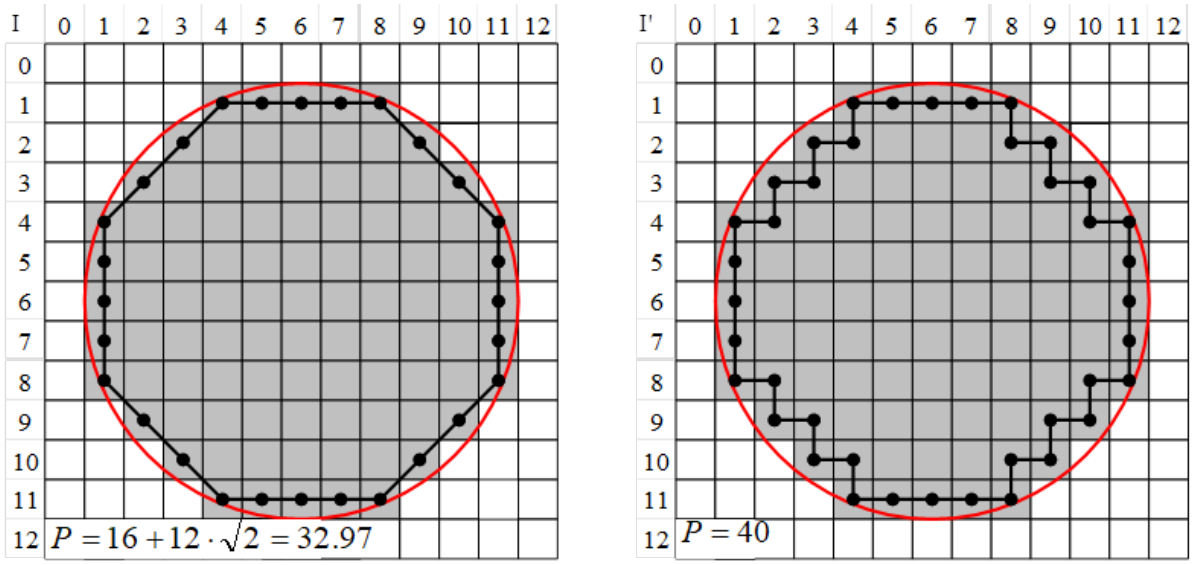



Figure 21.7. Calculation of the perimeter using an 8-connected neighborhood (left) and a 4-connected neighborhood (right)

## 21.3. VisualApplets Operators

VisualApplets offers two operators for performing a blob analysis. These are the "Blob_Analysis_2D" operator for ordinary 2D images, and the "Blob_Analysis_1D" operator for line camera applications that use endless image heights.

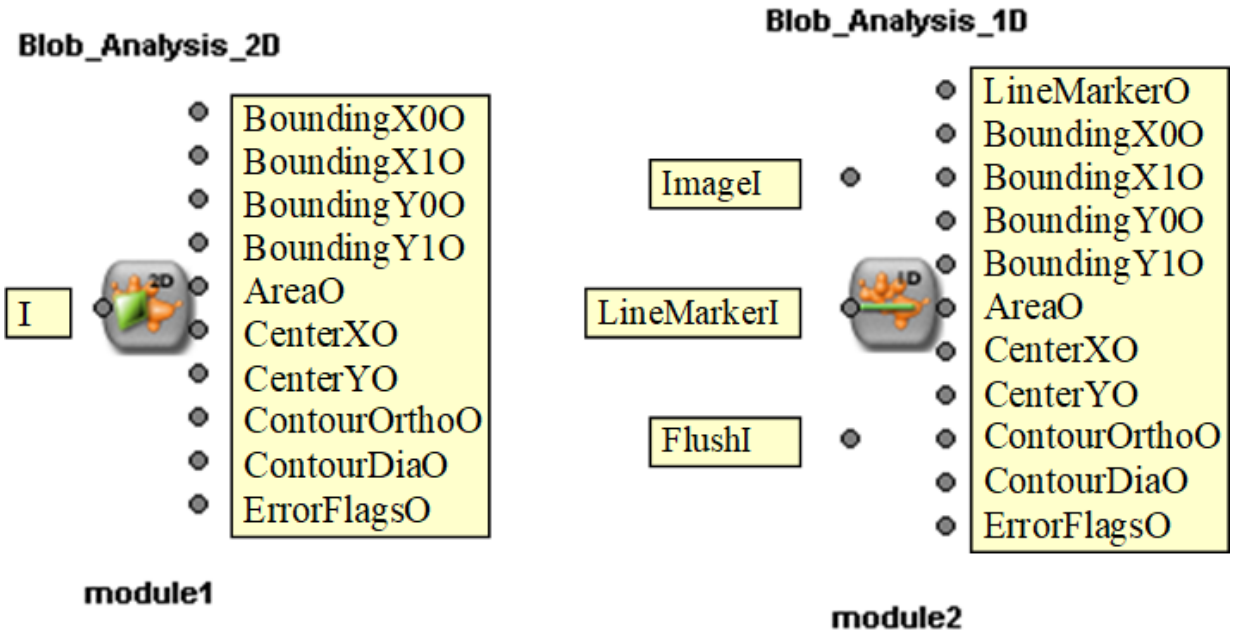



Figure 21.8. Blob Analysis Operators

The operator reference (below) provides detailed explanations for both operators.

The following list summarizes all Operators of Library Blob

| Operator Name | | Short Description | available since |
|---|---|---|---|
| | Blob_Analysis_1D | Detects objects in binary images of unlimited height and determines their properties. | Version 1.3 |
| | Blob_Analysis_2D | Detects objects in binary images and determines their properties. | Version 1.3 |

Table 21.1. Operators of Library Blob

## 21.4. Operator Blob_Analysis_1D

**Operator Library: Blob**

The Blob_Analysis_1D operator detects objects in binary images of infinite height and determines their properties. The outputs of the operator are several streams of data which represent properties for each object.



**You Might Need the 2D Operator Instead**

The Blob_Analysis_1D operator is used for acquisitions where objects are located in arbitrary positions. If in your line scan application your object positions can be determined with an image trigger, use the Blob_Analysis_2D operator instead.



**Availability**

To use the *Blob_Analysis_1D* operator, you need either a **Segmentation and Classification Library** license, or the **VisualApplets 4** license.

This operator reference manual includes the following information:

- Explanation of the operator's functionality
- Timing model for 1D applications
- Using the operator in the VisualApplets high level simulation
- Operator input and output ports
- I/O Properties
- Supported Link Formats
- Parameters
- Examples of Use

### 21.4.1. Explanation of the Operator's Functionality

For a general introduction into the blob analysis operators it is mandatory to read the introduction in 21. *Library Blob*.

The Blob Analysis 1D operator is designed for endless one-dimensional images. In general, for 2D Blob Analysis, the object features are related to the position of the object in the frame and the coordinate point of origin is the top left corner of the image. This is not possible for 1D images with endless height. The objects cannot be related to a y-coordinate as this value would grow infinitely or cause an overflow.

The VisualApplets Blob Analysis 1D operator allows the use of endless images where objects may have any position. The operator is extended by a reference input "LineMarkerI" which the objects are related to. This may be counter values controlled by an external encoder and trigger pulse, for example. Using this method, it is possible to allocate each object determined by the blob analysis to the correct reference position.

The input link "FlushI" is used to flush the output, i.e., complete an output frame. For example, this is required to complete a DMA transfer.

The following figure illustrates the behavior of the Blob Analysis 1D operator. It shows exemplified data at the input of the operator and the resulting output. The binary endless image processed by the Blob Analysis is shown on the left. The second column represents the LineMarkerI input. Here, a counter is used to mark the objects with reference values. For better visualization, only the values of interest are printed. The line marker input is assumed to have a bit width of nine bits. Hence, an overflow occurs at value 512. In the third column, the flush input is shown. In this example there is only one flush

condition. The right column represents the output of the Blob Analysis operator. Looking at the example you see that the image together with the line marker is processed during Blob Analysis. As soon as an object is completed, it is output. The resulting object properties of all objects are marked with the line marker which was set together with the first image line of the respective object. The bounding box coordinates X0 and X1 are similar to the Blob Analysis 2D object features, whereas the Y1 coordinate is the height of the object. The Y0 coordinate results of an internal counter of the Blob Analysis 1D operator. It starts with zero and its maximum value is equal to the maximum height of objects set in the property dialog of the operator. The flush condition causes an end of frame flag at the output.

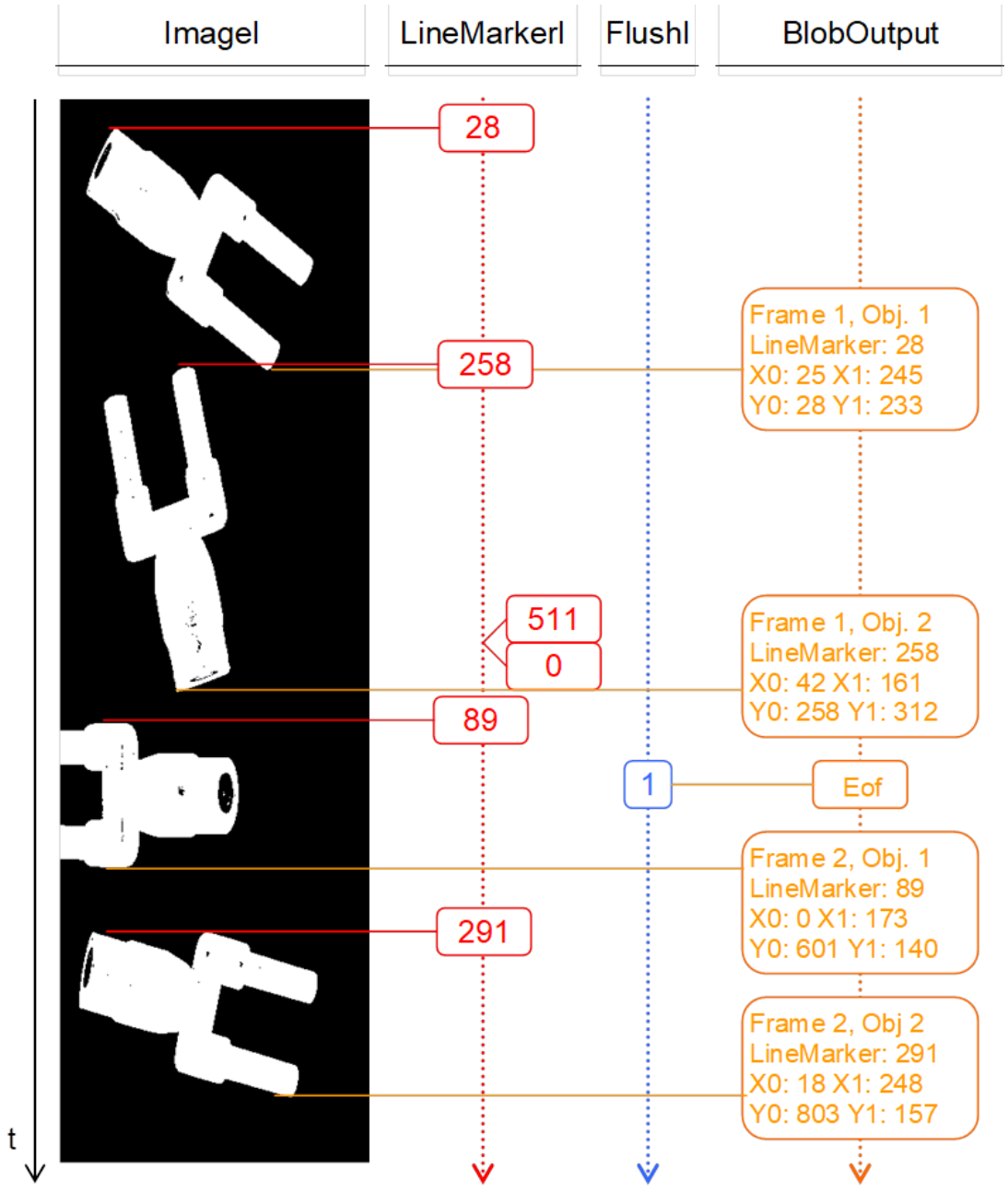



Figure 21.9. Behavior of the Blob Analysis 1D Operator

Similar to the 2D operator, the Blob Analysis 1D operator is of type "M" where all outputs are synchronous. In contrast to 2D, the 1D operator has three inputs. The first two inputs "ImageI" and "LineMarkerI" are synchronous and have to be sourced by the same M-type operator. The input link "FlushI" is asynchronous to the image and therefore does not need any synchronization. The following figure illustrates one possible configuration (amongst many others).

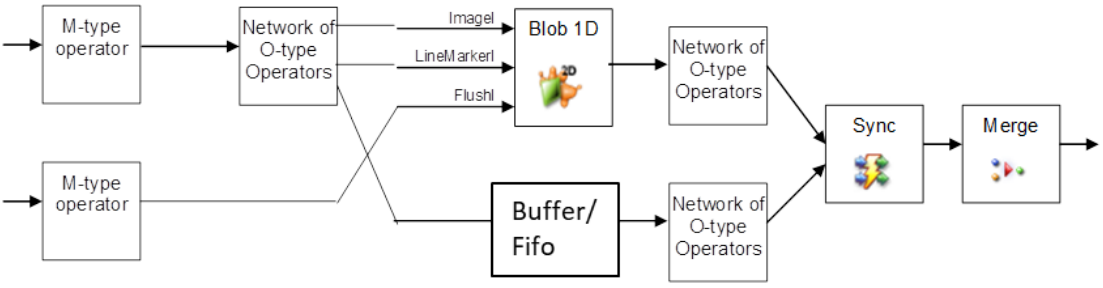



Figure 21.10. Synchronization of the Blob 1D Operator in a VisualApplets Network

## 21.4.2. Generation of Output Frames - Flush, Empty Frames, Discarding Data

As mentioned above, the flush is used to finalize an output frame, i.e., to generate the end of frame flag. Depending on the timing of the flush input and the objects in the image, it could be possible to generate empty frames as well as output frames containing a lot of data. To control this behavior, parameters *output_frame_size_overflow_handling* and *suppress_empty_output_frames* (see parameter description below) are used. The following waveforms show sketches of different combinations of parameter settings, data, and flush input. The "Blob Data" represents the output timing.

The first example shows the generation of new frames after the maximum frame size has been reached. The blob operator generates an end of frame after the last blob has been output which completed the output frame. Flush signals will generate additional ends of frames:

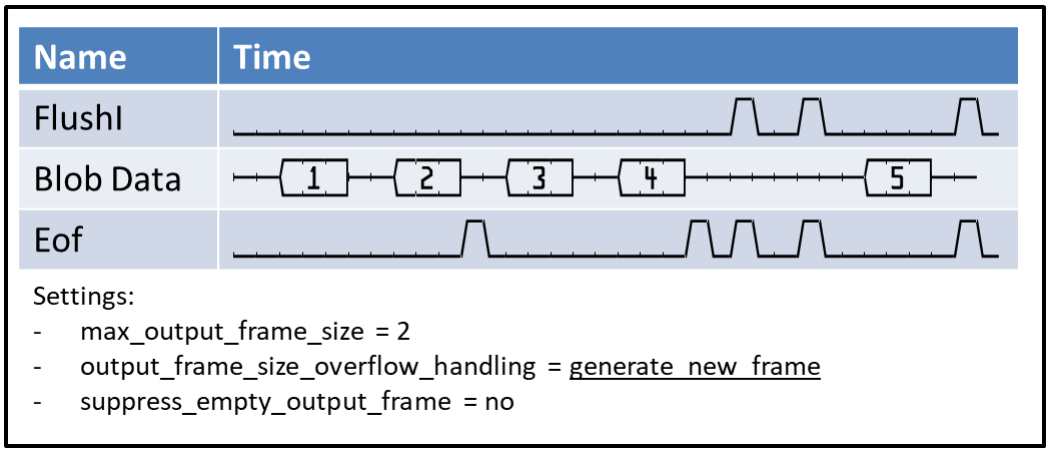



Figure 21.11. Blob 1D Timing - Generation of New Frames

The next example shows the suppression of empty output frames. As you see, the first flush is ignored. The second flush is delayed until a blob is available for output. This way, the operator can delay flush outputs:

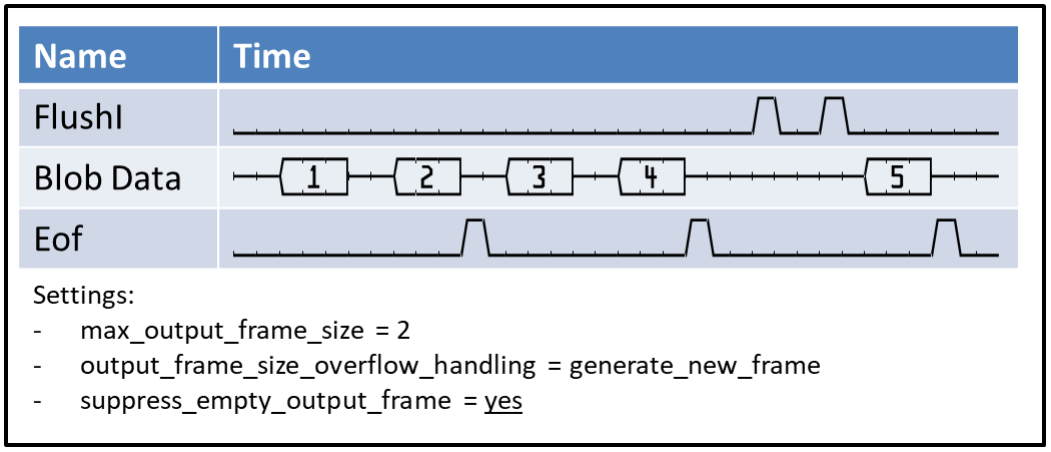



Figure 21.12. Blob 1D Timing - Suppression of Empty Frames

In the third example we se the behavior of a constant 1 at the flush input. As suppress_empty_output_frame is set to "yes", the operator will generate an end of frame after each blob output. Warning: If suppress_empty_output_frame is accidently set to "no", constant one can result in millions of ends of frames. If this output is connected to the VisualApplets output (DMA operator), it could cause a PC overload!

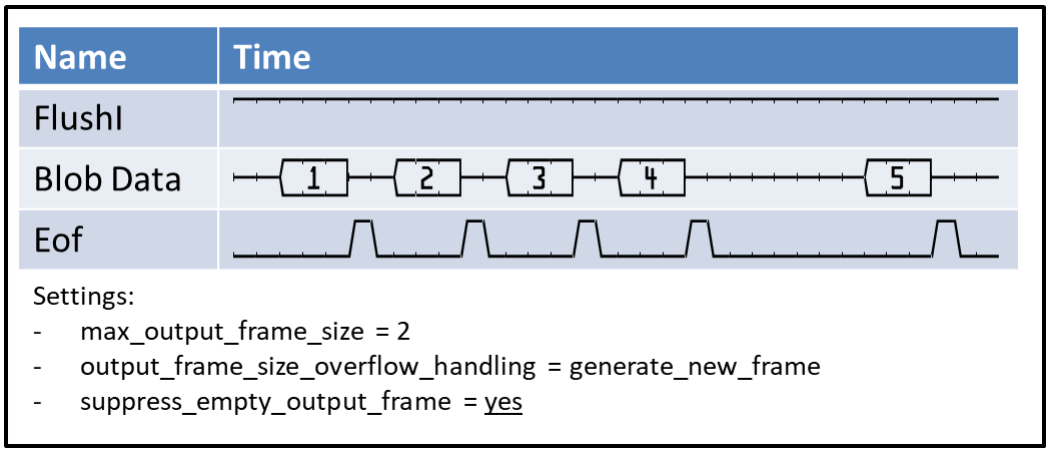



Figure 21.13. Blob 1D Timing - Constant Flush

The last example shows the discarding of objects, as parameter "output_frame_size_overflow_handling" is set to "discard_exceeding_objects".

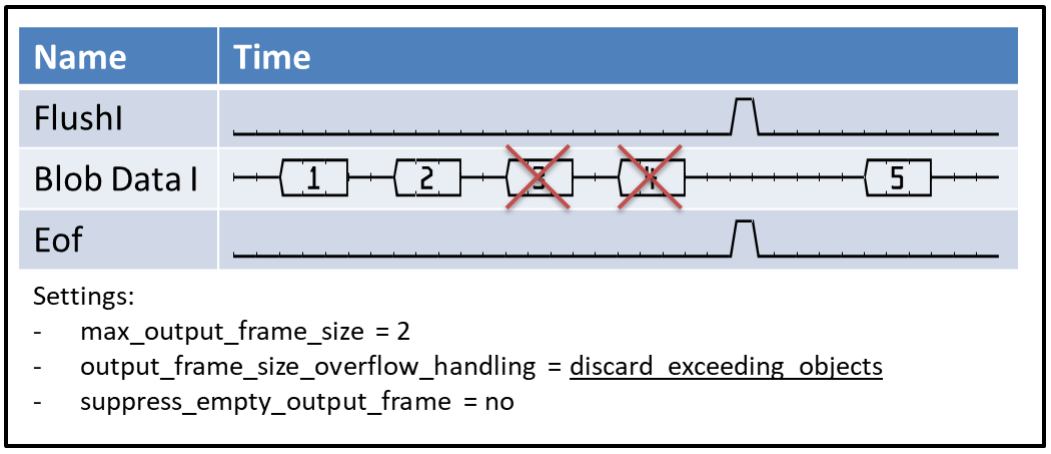



Figure 21.14. Blob 1D Timing - Discarding of Objects

Please note that the output timing depends on the output pipeline. If data cannot be processed, i.e., the successive operators behind the blob operator block the processing, it might happen that flush signals are lost.

### 21.4.3. Performance

The VisualApplets blob analysis operators are some of the very few operators where the processing speed, i.e., the bandwidth, depends on the image content. Many objects in an image cause a large object list what may result in slowing down the operator input. This applies only for images containing very strong noise. For controlled conditions, the operator should be sufficiently fast.

### 21.4.4. Latency

The operator is designed to have a latency reduced to the minimum. Input images are transferred to the Blob Analysis line by line. If an object is completely transferred into the operator, its object features are output immediately once the operator can detect its completion. Hence, the Blob Analysis outputs objects as soon as they are completely transferred to the operator. The post-processing of the object features can be started while the image itself is still being processed.

Note that the DMA and some other operators wait for the frame end signal before they report completion. Check the description of the flush input to learn about frame end generation in the Blob_Analysis_1D operator.

### 21.4.5. Simulation of the Operator



**Use the Simulation of Blob_Analysis_1D with Caution**

The Blob 1D simulation has some limitations. Please use the simulation feature with due caution.

VisualApplets uses normal 2D images, even if the image protocol is 1D. So for simulating the operator, we use normal images. Each image is treated individually.

The simulation of the operator is NOT equal to the hardware behavior. There are two differences:

- The order of the object feature output might differ in hardware and software. This is because the hardware output depends on the timing of the data which cannot be simulated in VisualApplets.
- The second (and most important) difference is the behavior of the Flush conditions. The FlushI input is completely asynchronous to the image data input. FlushI is used to complete the output frames, i.e., it inserts end-of-frame markers into the output stream. (See above, and parameter description below.) As the flush is asynchronous to the image data, it cannot be simulated to reflect the same behavior as in hardware. Therefore, the VisualApplets simulation uses an alternative simulation model. As the behavior of the hardware cannot be copied, the simulation model is implemented differently to the hardware on purpose. It has the following properties:
  - The simulation tries to compare the correlation of the data input and the flush input by counting the pixels of both inputs. So if the data input and flush input have the same image dimensions, the correlation is 1:1 to the pixel position. However, if the flush input has less pixel (for example 1 pixel per line), only the simulation will compare the pixel positions.
  - The flush condition will output an object in comparison to the Y0 coordinate of the object and not in comparison to its completion as done in hardware.
  - After the last object of an image has been output, all further flush signals are ignored. So in simulation, the number of flush signals at the input are not correlated to the number of frame outputs.

  Because of this behavior, the simulation should be used with caution.

The following figures show some examples on mini-images, explaining some special conditions of using the flush signal in simulation.

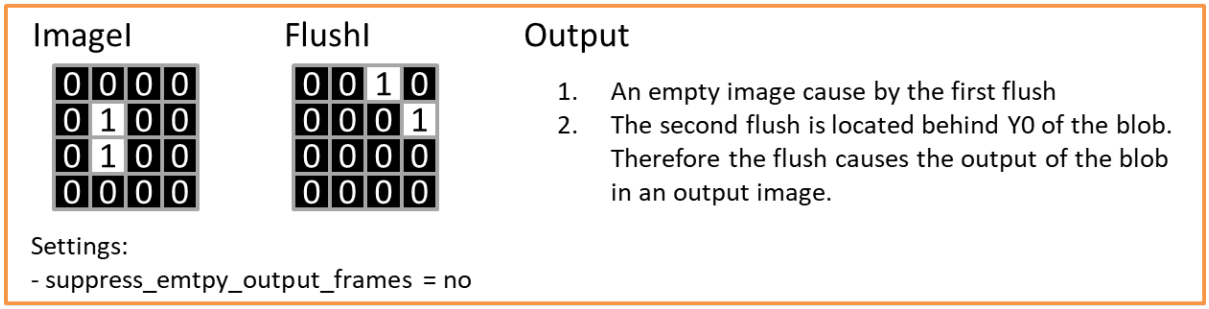



Figure 21.15. Simulation Scenario 1 - Flush and Y0 Relation

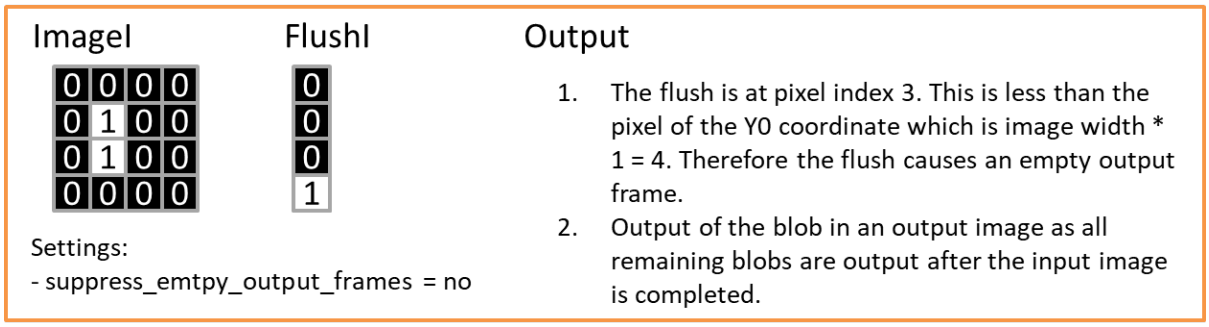



Figure 21.16. Simulation Scenario 2 - Flush Pixel Position

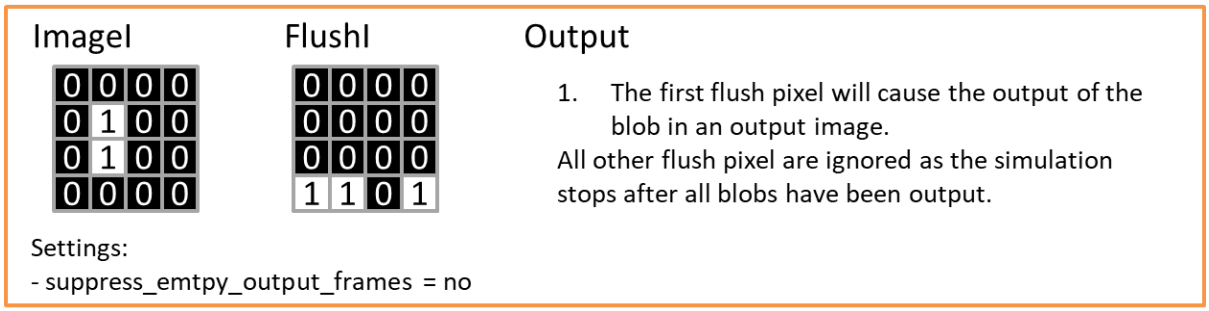



Figure 21.17. Simulation Scenario 3 - Discarded Flush Pixel at End of Frame

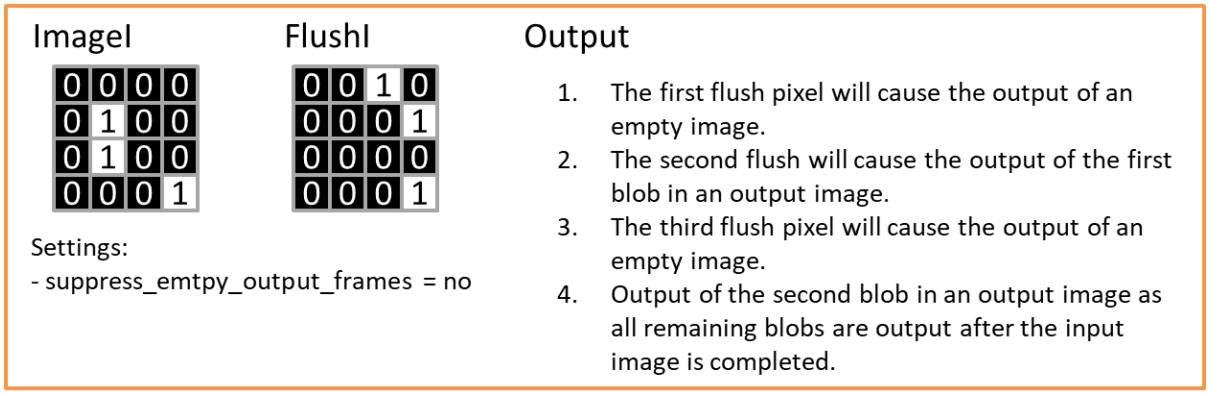



Figure 21.18. Simulation Scenario 4 - Multiple Blobs

### 21.4.6. Input Ports

The ImageI input of the Blob Analysis is represented by a binary one-dimensional image, i.e., an image having a bit with of 1Bit, a specified width, and an unlimited height. A parallelism of up to 32 pixels can be selected. Foreground values are assumed to have the value ONE, while background values must have the value ZERO. This must be considered at the binarization process.

As the LineMarkerI input is synchronous to the image data input, these links must have the same image dimensions and parallelism. However, the link may have any bit width. The value of LineMarker has to be constant throughout each image line.

The input link "FlushI" is asynchronous to the other input links. It may have any bit width, parallelism, and image dimensions. If one of the input bits of any of the parallel components is set, the Blob Analysis output is flushed.

### 21.4.7. Output Ports

Each of the output ports represents one object feature / object property. Each output value at these ports represents one object. Hence, the object properties result in a stream of data. The length of the output data streams is equal to the number of objects found in the image and can be interrupted into several sub-frames. Each output port has a height of 1 pixel and a specific length. They are represented as grayscale 2D images.

The output ports are configured by using the operator parameters and the properties of the input. The direct change of a link property is not possible.

The port ErrorFlagsO outputs several error flags of overflows. Each bit is reserved for a special flag. A detailed explanation can be found in the parameter description below. A summary is given in the following table:

| Bit # | Description | Object Related | Notes |
|---|---|---|---|
| 0, 1 | label overflow | no | The Blob Analysis has detected too many objects to store in memory. To increase the maximum number of objects within two image lines, operator parameter "LabelBits" can be changed.<br><br>The flag is set upon detection until the end of the data output frame. All object properties which have been output so far are valid. |

| Bit # | Description | Object Related | Notes |
|---|---|---|---|
| | | | For the Blob Analysis 1D operator, the flag has only high-level state for one clock cycle together with the next valid data output.<br><br>After the flag is set, the output of the blob analysis might result in wrong object properties. The operator returns to correct functionality if enough object labels are available. |
| 2 | output truncated | yes | The number of objects in the image exceed the maximum output width set in the operator parameters dialog. The flag is set together with the last object which fits into the output width. |
| 3 | object size exceeds maximum | yes | The flag is set if an object of the 1D Blob Analysis exceeds its maximum height. The operator will cut this exceeding object into smaller objects. The continuative object is marked with the flag to allow the detection of a cut.<br><br>For 2D Blob Analysis the flag is set to const zero. |
| 4 | area is truncated | yes | The area of the object together with the error flag is larger than the area bits allow. |
| 5 | center of gravity X is truncated | yes | The center of gravity in X-direction is larger than the bits allow. |
| 6 | center of gravity Y is truncated | yes | The center of gravity in Y-direction is larger than the bits allow. |
| 7 | contour length overflow | yes | The contour length is larger than the bits parameterized for the operator allow. |

Table 21.2. Explanation of Blob Error Flags

## 21.4.8. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | ImageI, image data input<br>LineMarkerI, data input<br>FlushI, data input |
| **Output Links** | LineMarkerO, data output<br>BoundingX0O, data output<br>BoundingX1O, data output<br>BoundingY0O, data output<br>BoundingY1O, data output<br>AreaO, data output<br>CenterXO, data output<br>CenterYO, data output<br>ContourOrthoO, data output<br>ContourDiaO, data output<br>ErrorFlagsO, data output |

**Synchronous and Asynchronous Inputs**

- Synchronous Group: *ImageI*, *LineMarkerI*
- *FlushI* is asynchronous to the group.

### 21.4.9. Supported Link Format

| Link Parameter | Input Link ImageI | Input Link LineMarkerI | Input Link FlushI |
|---|---|---|---|
| **Bit Width** | 1 | any | any |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | {1, 2, 4, 8, 16, 32} | as ImageI | any |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_IMAGE1D | VALT_IMAGE1D | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | any | as ImageI | any |
| **Max. Img Height** | any | as ImageI | any |

| Link Parameter | Output Link LineMarkerO | Output Link BoundingX0O | Output Link BoundingX1O |
|---|---|---|---|
| **Bit Width** | auto❶ | auto❷ | auto❸ |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_IMAGE1D | VALT_IMAGE1D | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | auto❹ | auto❺ | auto❻ |
| **Max. Img Height** | 1 | 1 | 1 |

| Link Parameter | Output Link BoundingY0O | Output Link BoundingY1O | Output Link AreaO |
|---|---|---|---|
| **Bit Width** | auto❼ | auto❽ | auto❾ |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_IMAGE1D | VALT_IMAGE1D | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | auto❿ | auto⓫ | auto⓬ |
| **Max. Img Height** | 1 | 1 | 1 |

| Link Parameter | Output Link CenterXO | Output Link CenterYO | Output Link ContourOrthoO |
|---|---|---|---|
| **Bit Width** | auto⓭ | auto⓮ | auto⓯ |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |

| Link Parameter | Output Link CenterXO | Output Link CenterYO | Output Link ContourOrthoO |
|---|---|---|---|
| **Img Protocol** | VALT_IMAGE1D | VALT_IMAGE1D | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | auto⓰ | auto⓱ | auto⓲ |
| **Max. Img Height** | 1 | 1 | 1 |

| Link Parameter | Output Link ContourDiaO | Output Link ErrorFlagsO |
|---|---|---|
| **Bit Width** | auto⓳ | 8 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_IMAGE1D | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | auto⓴ | auto㉑ |
| **Max. Img Height** | 1 | 1 |

❶❷❸❼❽❾⓭⓮⓯⓳ The output bit width depends on the parameter settings of the object feature. If a feature is disabled, the output is set to one bit which is always value zero.

❹❺❻❿⓫⓬⓰⓱⓲⓴㉑ The output maximum image width depends on the settings of parameter *output_frame_size*.

## 21.4.10. Parameters

| label_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | 7 |
| **Range** | [5, 31] |

This parameter sets the number of bits which are used to label the objects internally. It also influences the maximum number of objects which may coexist within two image lines determined by 2^label_bits. Note that the required memory resources for the blob analysis almost double with every additional bit. A good value range for this parameter is between 7 and 9 Bits which should be sufficient for almost every application.

If an overflow of this address space is detected, the operator outputs error flags at bits 0 and 1 of port "ErrorFlagsO". These flags are set upon detection of the overflow and are reset with the end of the output frame.

| neighborhood | |
|---|---|
| **Type** | static parameter |
| **Default** | eight_connected |
| **Range** | {four_connected, eight_connected} |

Select the required neighborhood for object detection. See Section 21.1, 'Definition' for a detailed explanation of pixel neighborhoods.

| max_object_height_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | 10 |

| max_object_height_bits | |
|---|---|
| **Range** | [3, 31] |
| This parameter is required to specify the maximum height of an object. For example, a height of 10 bits will not allow objects exceeding a height of 1024 image lines.<br><br>If an object exceeds this limitation, it is cut into two ore more separated objects. If this happens, an error flag at bit number 3 of port ErrorFlagsO is set together with the resumed object. | |

| max_output_frame_size | |
|---|---|
| **Type** | static parameter |
| **Default** | 65536 |
| **Range** | [1, 67108864] |
| Select the maximum output frame size, i.e., the maximum number of objects in one output frame. | |

| output_frame_size_overflow_handling | |
|---|---|
| **Type** | static parameter |
| **Default** | generate_new_frame |
| **Range** | {generate_new_frame, discard_exceeding_objects} |
| The maximum output frame size is specified with parameter "max_output_frame_size". If the number of objects exceeds this size, the operator can either generate a new frame or discard the exceeding objects. The generation of a new frame results in an end of frame, i.e., the frame is completed. If objects are discarded, the operator outputs an error flag at bit no. 2 of port ErrorFlagsO together with the last valid object of the frame.<br><br>Note that a new frame is also generated with the flush input. | |

| suppress_empty_output_frames | |
|---|---|
| **Type** | static parameter |
| **Default** | yes |
| **Range** | {yes, no} |
| This parameter is used to suppress empty output frames caused by a flush condition and no object data.<br><br>Detailed timing diagrams can be found in Section 21.4.2, 'Generation of Output Frames - Flush, Empty Frames, Discarding Data'. | |

| line_marker | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |
| The "line_marker" of each object is output at "LineMarkerO". The bit width is determined by the input LineMarkerI input link. If the line marker is not used, the bit width is set to one with constant zero at its output. | |

| bounding_box_x0 | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |
| A bounding box represents the minimum paraxial rectangle which fits over the object. Each coordinate value is relatively to the top left corner of the image. The bounding box X0 represents the lowest x-coordinate of each object. It is output at "BoundingX0O". The bit width is determined by the input image. If the bounding box is not used, the bit width is set to one bit with constant zero at its output. | |

| bounding_box_x1 | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |

A bounding box represents the minimum paraxial rectangle which fits over the object. Each coordinate value is relatively to the top left corner of the image. The bounding box X1 represents the highest x-coordinate of each object. It is output at "BoundingX1O". The bit width is determined from the input image. If the bounding box is not used, the bit width is set to one bit with constant zero at its output.

| bounding_box_y0 | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |

The bounding box y0 represents the lowest y-coordinate of each object. This coordinate is based on an internal counter of the operator. It starts counting with the first image line processed. Its maximum value is equal to the maximum height set with the parameter max_object_height_bits. After an overflow, the counter starts from zero again.

This feature can be used instead of the line marker. However, the line marker offers more possibilities and allows complex configurations.

The feature is output at "BoundingY0O". The bit width is determined by the input image. If the bounding box is not used, the bit width is set to one with constant zero at its output.

| bounding_box_y1 | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |

With the the 1D operator (in contrast to the 2D operator), bounding box y1 represents the height of an object. It is output at "BoundingY1O". The bit width is determined by the input image. If the bounding box is not used, the bit width is set to one with constant zero at its output.

| area_mode | |
|---|---|
| **Type** | static parameter |
| **Default** | use_area_with_maximum_required_bits |
| **Range** | {area_not_used, use_area_with_maximum_required_bits, use_area_with_specified_bits} |

The area of an object is defined by the sum of all object foreground pixels. This parameter is used to select the required area mode. If "use_area_with_maximum_required_bits" is selected, the operator will automatically determine the required bits for the maximum possible size of an object. The maximally possible size of an object dependeds on the maximum width and height at the input link I. The theoretical maximum of an object is achieved if all pixels of an image consist of foreground values, i.e., a white input image which is one large object.

If users can be sure that objects will not have a larger area than a specified value, this can be parameterized. Select "use_area_with_specified_bits" if the maximum object size is known.
Use parameter "area_bits" to specify the number of bits. If the area is not required at all, select "area_not_used". In "area_not_used" mode, the bit width is set to one with constant zero at its output.

| area_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | not_used |
| **Range** | {not_used, used} |

| area_bits |
|---|
| This parameter is enabled only if parameter "area_mode" is set to "use_area_with_specified_bits". If the area of an object is larger than the selected bits allow for, the blob analysis will output an overflow flag at bit 4 of the output link "ErrorFlagsO". |

| center_of_gravity_x_mode | |
|---|---|
| **Type** | static parameter |
| **Default** | use_cX_with_maximum_required_bits |
| **Range** | {cX_not_used, use_cX_with_maximum_required_bits, use_cX_with_specified_bits} |
| See Section 21.2.3, 'Center of Gravity' for an explantion of the Center of Gravity.<br><br>Note that the output has to be divided by the area after blob analysis to get correct results.<br><br>If the parameter is set to "use_cX_width_maximum_required_bits", the operator will determine the output bits automatically. The required number of bits can get very high if large input images are used.<br><br>The number of bits is adjustable if "use_cX_with_specified_bits" is used for parameter value.<br><br>If no center of gravity in x-direction is required, it can be switched off by selecting "cX_not_used". | |

| center_of_gravity_x_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | 29 |
| **Range** | [2, auto] |
| If "use_cX_with_specified_bits" is set for "center_of_gravity_x_mode", the bits can be changed here. Otherwise, the parameter is disabled. | |

| center_of_gravity_x_overflow_flag | |
|---|---|
| **Type** | static parameter |
| **Default** | not_used |
| **Range** | {not_used, used} |
| If the number of bits for the center of gravity in x-direction is set manually, this parameter is activated and can be set to "used". The operator will then output a ONE at bit index 5 of the output link "ErrorFlagsO" if an overflow is detected. | |

| center_of_gravity_y_mode | |
|---|---|
| **Type** | static parameter |
| **Default** | use_cY_with_maximum_required_bits |
| **Range** | {cY_not_used, use_cY_with_maximum_required_bits, use_cY_with_specified_bits} |
| See Section 21.2.3, 'Center of Gravity' for an explantion of the Center of Gravity.<br><br>Note that the output has to be divided by the area after blob analysis to get correct results.<br><br>If the parameter is set to "use_cY_width_maximum_required_bits", the operator will determine the output bits automatically. The required number of bits can get very high if large input images are used.<br><br>The number of bits is adjustable if "use_cY_with_specified_bits" is used for parameter value.<br><br>If no center of gravity in y-direction is required, it can be switched off by selecting "cY_not_used". | |

| center_of_gravity_y_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | 29 |

| center_of_gravity_y_bits | |
|---|---|
| **Range** | [2, auto] |
| If "use_cY_with_specified_bits" is set for "center_of_gravity_y_mode", the bits can be changed here. Otherwise, the parameter is disabled. | |

| center_of_gravity_y_overflow_flag | |
|---|---|
| **Type** | static parameter |
| **Default** | not_used |
| **Range** | {not_used, used} |
| If the number of bits for the center of gravity in y-direction is set manually, this parameter is activated and can be set to "used". The operator will then output a ONE at bit index 6 of the output link "ErrorFlagsO" if an overflow is detected. | |

| contour_length_mode | |
|---|---|
| **Type** | static parameter |
| **Default** | contour_length_not_used |
| **Range** | {contour_length_not_used, contour_length_used} |
| The contour length of an object includes all edges, even at holes. For a detailed explanation see Section 21.2.4, 'Contour Length'.<br><br>If this feature is selected, the required bits can be chosen with the parameters "contour_length_bits_orthogonal_connected" and "contour_length_bits_diagonal_connected" depending on the neighborhood selected. If a 4-connected neighborhood is selected, the contour length can only be determined in orthogonal directions. If an 8-connected neighborhood is selected the contour length is determined in orthogonal and diagonal directions. | |

| contour_length_bits_orthogonal_connected | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1, 31] |
| If "contour_length_mode" is set to "contour_length_used", the bits required to represent the contour length for orthogonal connected pixels can be chosen here. | |

| contour_length_bits_diagonal_connected | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1, 31] |
| If "contour_length_mode" is set to "contour_length_used", the bits required to represent the contour length for diagonal connected pixels can be chosen here. This is only possible if an 8-connceted neighborhood is selected. Otherwise, this parameter is disabled. | |

| contour_length_overflow_flag | |
|---|---|
| **Type** | static parameter |
| **Default** | not_used |
| **Range** | {not_used, used} |
| The contour length may cause an overflow. If the flag is used, an error flag is set at bit number 7 of port "ErrorFlagsO" in case of an overflow. | |

## 21.4.11. Examples of Use

The use of operator Blob_Analysis_1D is shown in the following examples:

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

# 21.5. Operator Blob_Analysis_2D

**Operator Library: Blob**

The Blob_Analysis_2D operator detects objects in binary images and determines their properties. The outputs of the operator are several streams of data which represent properties for each object.

The functionality of the operator can be fully simulated in VisualApplets. However, please note that the order of the output values might not be equal to the hardware implementation.



**Availability**

To use the *Blob_Analysis_2D* operator, you need either a **Segmentation and Classification Library** license, or the **VisualApplets 4** license.

**Make sure you first read the general introduction into the blob analysis operators (section 21. *Library Blob*).**

## 21.5.1. Performance

The VisualApplets blob analysis operators belong to the few operators whose processing speed, i.e., bandwidth, depends on the image content. Are many objects in the image, a large object list is created. This may result in slowing down the operator input. However, this applies only for images containing very strong noise. For controlled conditions, the operator should be sufficiently fast.

## 21.5.2. Latency

The operator works with minimum latency. Input images are transferred to the blob analysis line by line. As soon as an object has been transferred into the operator completely, the operator detects object completion and outputs the object features. Thus, post-processing of object features can be started while the image itself is still being processed by the operator.

Note that the DMA and some other operators wait for the frame end signal before they report completion. The frame end signal is generated once the input image is finalized.

## 21.5.3. Input Ports

The ImageI input of the blob analysis is represented by a binary, two-dimensional image, that is, an image having a bit width of 1 Bit, a specified image width, and a specified image height. You can select a parallelism of up to 32 pixels.

The operator assumes foreground values to have value ONE, and background values to have value ZERO. Make sure you match this requirement in the preliminary binarization process.

## 21.5.4. Output Ports

Each of the output ports represents one object feature / object property. Each output value at these ports represents one object. Hence, each object property results in one data stream. The length of the output data streams is equal to the number of objects found in the image. The streams can be interrupted into several sub-frames. The data on each output port has a height of 1 pixel and a specific length. The output is represented in form of grayscale 2D images.

The output ports are configured using the operator parameters and properties of the input. The direct change of a link property is not possible.

The port ErrorFlagsO outputs several error flags of overflows. Each bit is reserved for a special flag. A detailed explanation can be found in the parameter description below. A summary is given in the following table:

| Bit # | Description | Object Related | Notes |
|---|---|---|---|
| 0, 1 | label overflow | no | The Blob Analysis has detected too many objects to store in memory. To increase the maximum number of objects within two image lines, operator parameter "LabelBits" can be changed.<br><br>The flag is set upon detection until the end of the data output frame. All object properties which have been output so far are valid.<br><br>For the Blob Analysis 1D operator, the flag has only high-level state for one clock cycle together with the next valid data output.<br><br>After the flag is set, the output of the blob analysis might result in wrong object properties. The operator returns to correct functionality if enough object labels are available. |
| 2 | output truncated | yes | The number of objects in the image exceed the maximum output width set in the operator parameters dialog. The flag is set together with the last object which fits into the output width. |
| 3 | object size exceeds maximum | yes | The flag is set if an object of the 1D Blob Analysis exceeds its maximum height. The operator will cut this exceeding object into smaller objects. The continuative object is marked with the flag to allow the detection of a cut.<br><br>For 2D Blob Analysis the flag is set to const zero. |
| 4 | area is truncated | yes | The area of the object together with the error flag is larger than the area bits allow. |
| 5 | center of gravity X is truncated | yes | The center of gravity in X-direction is larger than the bits allow. |
| 6 | center of gravity Y is truncated | yes | The center of gravity in Y-direction is larger than the bits allow. |
| 7 | contour length overflow | yes | The contour length is larger than the bits parameterized for the operator allow. |

Table 21.3. Explanation of Blob Error Flags

### 21.5.5. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Links** | BoundingX0O, data output<br>BoundingX1O, data output<br>BoundingY0O, data output<br>BoundingY1O, data output<br>AreaO, data output<br>CenterXO, data output<br>CenterYO, data output<br>ContourOrthoO, data output<br>ContourDiaO, data output<br>ErrorFlagsO, data output |

### 21.5.6. Supported Link Format

| Link Parameter | Input Link I | Output Link BoundingX0O | Output Link BoundingX1O |
|---|---|---|---|
| **Bit Width** | 1 | auto❶ | auto❷ |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | {1, 2, 4, 8, 16, 32} | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | VALT_IMAGE2D |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | any | auto❸ | auto❹ |
| **Max. Img Height** | any | 1 | 1 |

| Link Parameter | Output Link BoundingY0O | Output Link BoundingY1O | Output Link AreaO |
|---|---|---|---|
| **Bit Width** | auto❺ | auto❻ | auto❼ |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | VALT_IMAGE2D |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | auto❽ | auto❾ | auto❿ |
| **Max. Img Height** | 1 | 1 | 1 |

| Link Parameter | Output Link CenterXO | Output Link CenterYO | Output Link ContourOrthoO |
|---|---|---|---|
| **Bit Width** | auto⓫ | auto⓬ | auto⓭ |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | VALT_IMAGE2D |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | auto⓮ | auto⓯ | auto⓰ |
| **Max. Img Height** | 1 | 1 | 1 |

| Link Parameter | Output Link ContourDiaO | Output Link ErrorFlagsO |
|---|---|---|
| **Bit Width** | auto⓱ | 8 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D |

| Link Parameter | Output Link ContourDiaO | Output Link ErrorFlagsO |
|---|---|---|
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | auto⓲ | auto⓳ |
| **Max. Img Height** | 1 | 1 |

❶❷❽❻❼⓫⓬⓭⓱
The output bit width depends on the parameter settings of the object feature. If a feature is disabled, the output is set to one bit which is always value zero.

❸❹❽❾❿⓮⓯⓰⓲⓳
The output maximum image width depends on the settings of parameter *output_frame_size*.

## 21.5.7. Parameters

| label_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | 7 |
| **Range** | [5, 31] |
| This parameter sets the number of bits which are used to label the objects internally. It also influences the maximum number of objects which may coexist within two image lines determined by 2^label_bits. Note that the required memory resources for the blob analysis almost double with every additional bit. A good value range for this parameter is between 7 and 9 bits which should be sufficient for almost every application.<br><br>If an overflow of this address space is detected, the operator outputs error flags at bits 0 and 1 of port "ErrorFlagsO". These flags are set upon detection of the overflow and are reset with the end of the output frame. | |

| neighborhood | |
|---|---|
| **Type** | static parameter |
| **Default** | eight_connected |
| **Range** | {four_connected, eight_connected} |
| Select the required neighborhood for object detection. See Section 21.1, 'Definition' for a detailed explanation of pixel neighborhoods. | |

| output_frame_size | |
|---|---|
| **Type** | static parameter |
| **Default** | specify_max_no_of_objects |
| **Range** | {maximum_required, specify_max_no_of_objects} |
| The maximum number of objects which may theoretically populate an image is: Input image width * input image height / 2 (equal to a check pattern using 4-connected neighborhood). This pattern is quite unlikely. Therefore, you can configure the maximum number of objects in the images via this parameter. | |

| max_output_frame_size | |
|---|---|
| **Type** | static parameter |
| **Default** | 65536 |
| **Range** | [1, 67108864] |
| Select the maximum number of objects at the output. Any more objects detected by the blob analysis will be discarded. However, on port "ErrorFlagsO", an error flag at bit 2 will be set. The flag is set together with the last valid object. This parameter is disabled if "ouput_frame_size" is set to "maximum_required". | |

| bounding_box_x0 | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |

A bounding box represents the minimum paraxial rectangle which fits over the object. Each coordinate value is relatively to the top left corner of the image. The bounding box X0 represents the lowest x-coordinate of each object. It is output at "BoundingX0O". The bit width is determined from the input image. If the bounding box is not used, the bit width is set to one bit with constant zero at its output.

| bounding_box_x1 | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |

A bounding box represents the minimum paraxial rectangle which fits over the object. Each coordinate value is relatively to the top left corner of the image. The bounding box X1 represents the highest x-coordinate of each object. It is output at "BoundingX1O". The bit width is determined from the input image. If the bounding box is not used, the bit width is set to one bit with constant zero at its output.

| bounding_box_y0 | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |

A bounding box represents the minimum paraxial rectangle which fits over the object. Each coordinate value is relatively to the top left corner of the image. The bounding box Y0 represents the lowest y-coordinate of each object. It is output at "BoundingY0O". The bit width is determined from the input image. If the bounding box is not used, the bit width is set to one with constant zero at its output.

| bounding_box_y1 | |
|---|---|
| **Type** | static parameter |
| **Default** | used |
| **Range** | {used, not_used} |

A bounding box represents the minimum paraxial rectangle which fits over the object. Each coordinate value is relatively to the top left corner of the image. The bounding box Y1 represents the highest y-coordinate of each object. It is output at "BoundingY1O". The bit width is determined from the input image. If the bounding box is not used, the bit width is set to one with constant zero at its output.

| area_mode | |
|---|---|
| **Type** | static parameter |
| **Default** | use_area_with_maximum_required_bits |
| **Range** | {area_not_used, use_area_with_maximum_required_bits, use_area_with_specified_bits} |

The area of an object is defined by the sum of all object foreground pixels. This parameter is used to select the required area mode. If "use_area_with_maximum_required_bits" is selected, the operator will automatically determine the required bits for the maximally possible size of an object. The maximally possible size of an object dependeds on the maximum width and height at the input link I. The theoretical maximum of an object is achieved if all pixels of an image consist of foreground values, i.e., a white input image which is one large object.

If users can be sure that their objects will not have a larger area than a specified value, this can be parameterized. Select "use_area_with_specified_bits" if the maximum object size is known.

| area_mode |
|---|
| Use parameter "area_bits" to specify the number of bits. If the area is not required at all, select "area_not_used". In "area_not_used" mode, the bit width is set to one with constant zero at its output. |

| area_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | not_used |
| **Range** | {not_used, used} |
| This parameter is enabled only if parameter "area_mode" is set to "use_area_with_specified_bits". If the area of an object is larger than the selected bits allow for, the blob analysis will output an overflow flag at bit 4 of the output link "ErrorFlagsO". | |

| center_of_gravity_x_mode | |
|---|---|
| **Type** | static parameter |
| **Default** | use_cX_with_maximum_required_bits |
| **Range** | {cX_not_used, use_cX_with_maximum_required_bits, use_cX_with_specified_bits} |
| See Section 21.2.3, 'Center of Gravity' for an explantion of the Center of Gravity.<br><br>Note that the output has to be divided by the area after blob analysis to get correct results.<br><br>If the parameter is set to "use_cX_width_maximum_required_bits", the operator will determine the output bits automatically. The required number of bits can get very high if large input images are used.<br><br>The number of bits is adjustable if "use_cX_with_specified_bits" is used for parameter value.<br><br>If no center of gravity in x-direction is required, it can be switched off by selecting "cX_not_used". | |

| center_of_gravity_x_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | 29 |
| **Range** | [2, auto] |
| If "use_cX_with_specified_bits" is set for "center_of_gravity_x_mode", the bits can be changed here. Otherwise, the parameter is disabled. | |

| center_of_gravity_x_overflow_flag | |
|---|---|
| **Type** | static parameter |
| **Default** | not_used |
| **Range** | {not_used, used} |
| If the number of bits for the center of gravity in x-direction is set manually, this parameter is activated and can be set to "used". The operator will then output a ONE at bit index 5 of the output link "ErrorFlagsO" if an overflow is detected. | |

| center_of_gravity_y_mode | |
|---|---|
| **Type** | static parameter |
| **Default** | use_cY_with_maximum_required_bits |
| **Range** | {cY_not_used, use_cY_with_maximum_required_bits, use_cY_with_specified_bits} |
| See Section 21.2.3, 'Center of Gravity' for an explantion of the Center of Gravity.<br><br>Note that the output has to be divided by the area after blob analysis to get correct results.<br><br>If the parameter is set to "use_cY_width_maximum_required_bits", the operator will determine the output bits automatically. The required number of bits can get very high if large input images are used. | |

| center_of_gravity_y_mode |
|---|
| The number of bits is adjustable if "use_cY_with_specified_bits" is used for parameter value.<br><br>If no center of gravity in y-direction is required, it can be switched off by selecting "cY_not_used". |

| center_of_gravity_y_bits | |
|---|---|
| **Type** | static parameter |
| **Default** | 29 |
| **Range** | [2, auto] |
| If "use_cY_with_specified_bits" is set for "center_of_gravity_y_mode", the bits can be changed here. Otherwise, the parameter is disabled. | |

| center_of_gravity_y_overflow_flag | |
|---|---|
| **Type** | static parameter |
| **Default** | not_used |
| **Range** | {not_used, used} |
| If the number of bits for the center of gravity in y-direction is set manually, this parameter is activated and can be set to "used". The operator will then output a ONE at bit index 6 of the output link "ErrorFlagsO" if an overflow is detected. | |

| contour_length_mode | |
|---|---|
| **Type** | static parameter |
| **Default** | contour_length_not_used |
| **Range** | {contour_length_not_used, contour_length_used} |
| The contour length of an object includes all edges, even at holes. For a detailed explanation see Section 21.2.4, 'Contour Length'.<br><br>If this feature is selected, the required bits can be chosen with the parameters "contour_length_bits_orthogonal_connected" and "contour_length_bits_diagonal_connected" depending on the selected neighborhood. If a 4-connected neighborhood is selected, the contour length can only be determined in orthogonal directions. If an 8-connected neighborhood is selected, the contour length is determined in orthogonal and diagonal directions. | |

| contour_length_bits_orthogonal_connected | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1, 31] |
| If "contour_length_mode" is set to "contour_length_used", the bits required to represent the contour length for orthogonally connected pixels can be chosen here. | |

| contour_length_bits_diagonal_connected | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1, 31] |
| If "contour_length_mode" is set to "contour_length_used", the bits required to represent the contour length for diagonally connected pixels can be chosen here. This is only possible if an 8-connceted neighborhood is selected. Otherwise this parameter is disabled. | |

| contour_length_overflow_flag | |
|---|---|
| **Type** | static parameter |
| **Default** | not_used |
| **Range** | {not_used, used} |

| contour_length_overflow_flag |
| --- |
| The contour length may cause an overflow. If the flag is used, an error flag is set at bit number 7 of port "ErrorFlagsO" in case of an overflow. |

## 21.5.8. Examples of Use

The use of operator Blob_Analysis_2D is shown in the following examples:

- Section 11.1, 'Functional Example for the *FrameBufferMultRoiDyn* Operator on the imaFlex CXP-12 Penta Platform'

  Examples - Demonstration of how to use the operator

- Section 12.3.2, 'Blob 2D'

  Examples - Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels.

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.13.2, 'Print Inspection Example- Position Correction and Defect Detection Using Blob Based Template Matching'

  Examples- Geometric Transformation and Defect Detection

- Section 12.13.3, 'Print Inspection Example- Position Correction and Defect Detection Using Image Moments and Blob Based Template Matching '

  Examples- Geometric Transformation and Defect Detection Using Image Moments

# 22. Library Color




The *Color* library includes operators for color space transformations and color processing.

The following list summarizes all Operators of Library Color

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | BAYER3x3Linear | Reconstructs an RGB image. Input link is a 3x3 kernel of a Bayer image. | Version 1.1 |
|  | BAYER5x5Linear | Reconstructs an RGB image. Input link is a 5x5 kernel of a Bayer image. | Version 1.1 |
|   | ColorTransform | Implements a matrix multiplication of a 3 by 3 matrix with an RGB vector. | Version 1.2 |
|   | HSI2RGB | Converts the color space from HSL (Hue Saturation Lightness) to RGB. | Version 1.1 |
|   | RGB2HSI | Converts the color space from RGB to HSL (Hue Saturation Luminance). | Version 1.1 |
|   | RGB2YUV | Converts the color space from RGB to YCbCr. | Version 1.1 |
|  | WhiteBalance | Scales the three color components of an RGB input stream. | Version 1.1 |
|  | WhiteBalanceBayer | Scales the three color components of a Bayer input stream. | Version 1.1 |

Table 22.1. Operators of Library Color

# 22.1. Operator BAYER3x3Linear

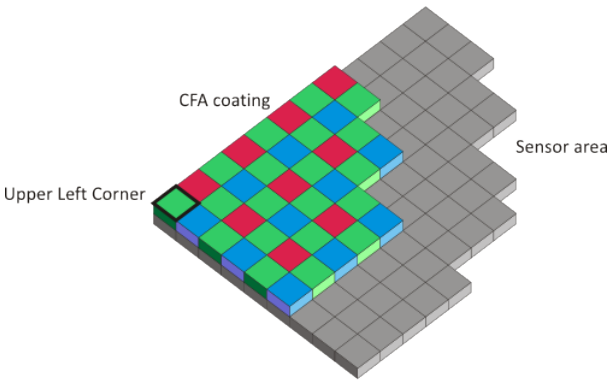
**Operator Library: Color**

The operator BAYER3x3Linear reconstructs an RGB image from a Bayer input image. Technically, a Bayer raw image is a grayscale image which is composed from alternating red, green and blue pixels. BAYER3x3Linear acquires an RGB image from a Bayer camera through a bilinear reconstruction. The input link must be a 3x3 kernel of a Bayer image.

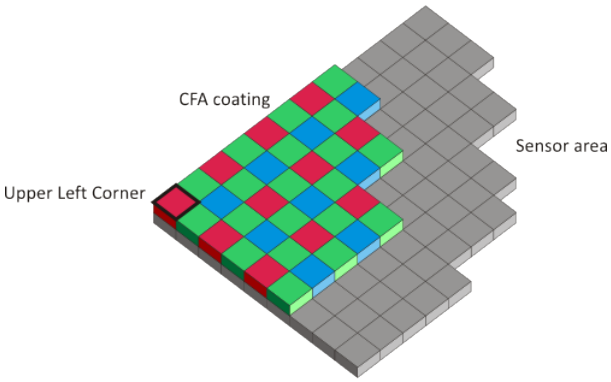
The operator input is a grayscale image while the output is a color image. Therefore, the output bit width is increased by a factor of three plus 2 precision bits per component. For example, an 8 bit grayscale input image will be reconstructed into a 3 * (8 + 2) bit color image. If the added precision bits are not required they can be discarded using operator *RND* or *ShiftRight*.

The Bayer array configuration of the camera must match the operator settings. The operator setting is changed using parameter *BayerInit*. Four different mappings of the Bayer array on the sensor are possible:

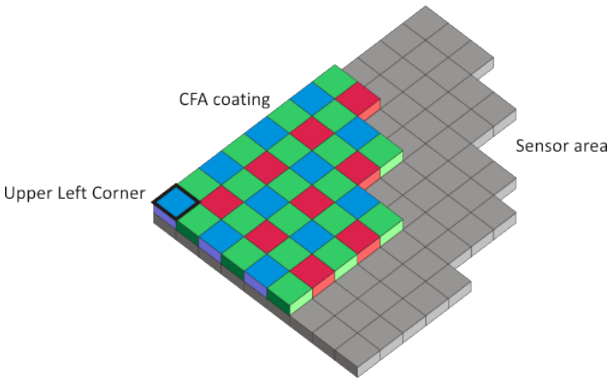
- **`GreenFollowedByRed`**



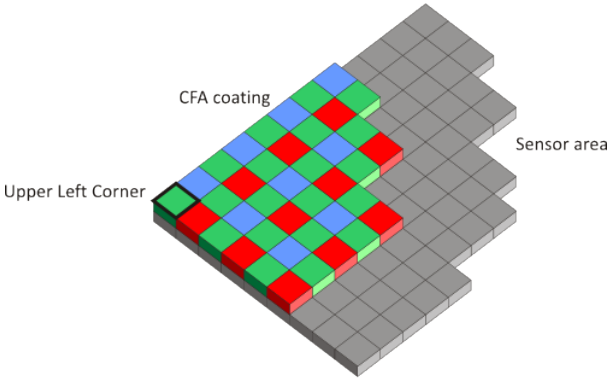

- **`RedFollowedByGreen`**




- **`BlueFollowedByGreen`**




- **`GreenFollowedByBlue`**




The figures show the top left corner of an image transferred to the frame grabber. In general, this is the top left corner of the camera's sensor but could change if you set a region of interest in the camera. The sensor pixel are overlaid with one of the four different Bayer patterns shown in the figures. Select the Bayer pattern corresponding to the image sensor of your camera.

The underlying algorithm is patent free.

### 22.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data input |

### 22.1.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 19] | (InputBitWidth + 2) * 3 |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 3 | 1 |
| **Kernel Rows** | 3 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | VAF_GRAY | VAF_COLOR |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Color Flavor** | FL_NONE | FL_RGB |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 22.1.3. Parameters

| BayerInit | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | GreenFollowedByRed |
| **Range** | {GreenFollowedByRed, GreenFollowedByBlue, RedFollowedByGreen, BlueFollowedByGreen} |
| The Bayer array configuration of the camera match the operator settings. See descriptions above. | |

### 22.1.4. Examples of Use

The use of operator BAYER3x3Linear is shown in the following examples:

- Section 12.4.1.2, 'Bayer 3x3 De-Mosaicing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter.

- Section 12.4.1.4, 'Bayer 3x3 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Moreover, a white balancing for color correction is added.

## 22.2. Operator BAYER5x5Linear

**Operator Library: Color**

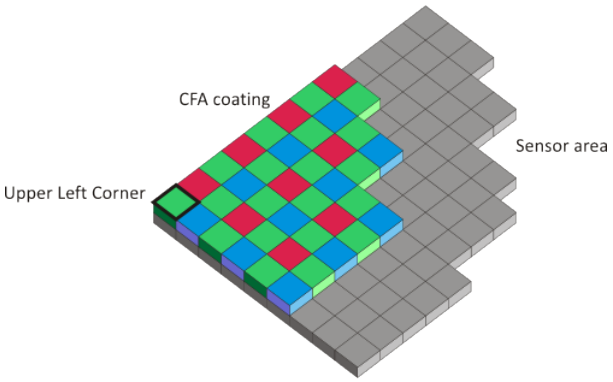
The operator BAYER5x5Linear reconstructs an RGB image from a Bayer input image. Technically, a Bayer raw image is a grayscale image which is composed from alternating red, green and blue pixels. BAYER5x5Linear acquires an RGB image from a Bayer camera through a bilinear reconstruction. The input link must be a 5x5 kernel of a Bayer image.

The underlying algorithm is a patent free high quality linear implementation.

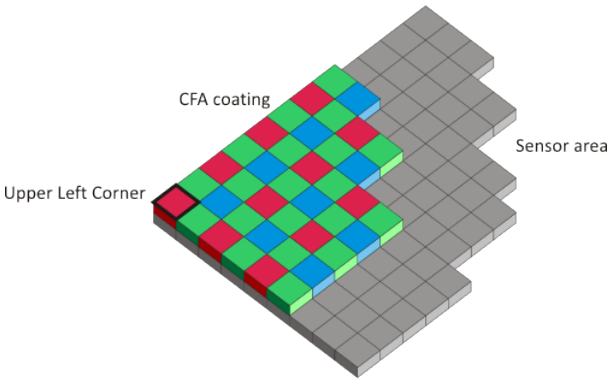
The operator input is a grayscale image while the output is a color image. Therefore, the output bit width is increased by a factor of three plus 4 precision bits per component. For example, an 8 bit grayscale input image will be reconstructed into a 3 * (8 + 4) bit color image. If the added precision bits are not required they can be discarded using operator *RND* or *ShiftRight*.

The Bayer array configuration of the camera must match the operator settings. The operator setting is changed using parameter *BayerInit*. Four different mappings of the Bayer array on the sensor are possible:

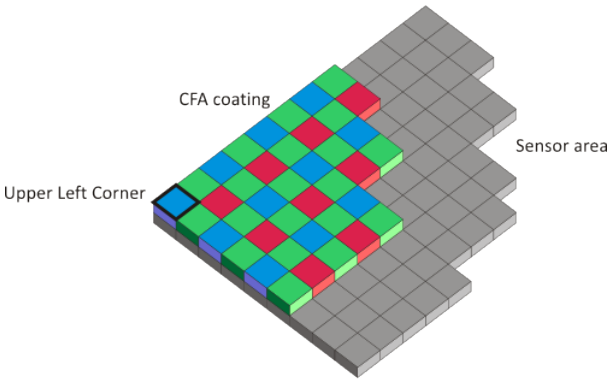
- **`GreenFollowedByRed`**



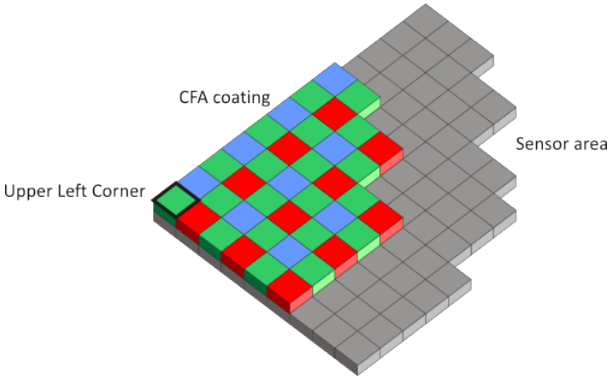

- **`RedFollowedByGreen`**




- **`BlueFollowedByGreen`**




- **`GreenFollowedByBlue`**




The figures show the top left corner an image transferred to the frame grabber. In general, this is the top left corner of the camera's sensor but could change if you set a region of interest in the camera. The sensor pixel are overlaid with one of the four different Bayer patterns shown in the figures. Select the Bayer pattern corresponding to the image sensor of your camera.

### 22.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data input |

### 22.2.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 16] | (InputBitWidth + 4) * 3 |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 3 | 1 |
| **Kernel Rows** | 3 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | VAF_GRAY | VAF_COLOR |
| **Color Flavor** | FL_NONE | FL_RGB |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 22.2.3. Parameters

| **BayerInit** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | GreenFollowedByRed |
| **Range** | {GreenFollowedByRed, GreenFollowedByBlue, RedFollowedByGreen, BlueFollowedByGreen} |
| The Bayer array configuration of the camera must match the operator settings. See descriptions above. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType influences the implementation strategy of the operator, i.e., which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy is selected automatically based on the parametrization of the connected links.

**EmbeddedALU:** The operator uses embedded arithmetic logic elements of the FPGA that are not LUT based.

**LUT:** The operator uses the LUT logic of the FPGA.



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

### 22.2.4. Examples of Use

The use of operator BAYER5x5Linear is shown in the following examples:

- Section 12.4.1.3, 'Bayer 5x5 De-Mosaicing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter.

- Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.9, 'Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern Red/BlueFollowedByGreen GreenFollowedByBlue/Red '

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a bilinear line scan camera with color pattern Red/BlueFollowedByGreen_GreenFollowedByBlue/Red

- Section 12.4.1.10, 'Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern RedFollowedByBlue GreenFollowedByGreen '

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a bilinear line scan camera with color pattern Red/BlueFollowedByBlue/Red_GreenFollowedByGreen

- Section 12.4.1.11, 'Bayer DeMosaicing a Line Scan Camera with Bi-color Bayer Pattern biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a CXP-12 line scan camera with bi-color Bayer pattern: biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR, for example the racer 2 L camera. In addition, the example contains a line scan trigger module and a white balancing module.

# 22.3. Operator ColorTransform

**Operator Library: Color**

The operator ColorTransform transforms the RGB stream at the input into an RGB stream with weighted components. The transformation is performed by a 3x3 matrix multiplication with the input RGB vector.

The transformation matrix is defined by the parameter *Coefficients*. This parameter is a 3x3 matrix containing double precision coefficients. The following formula represents the transformation algorithm.

$$\begin{pmatrix} R_o \\ G_o \\ B_o \end{pmatrix} = \begin{pmatrix} C0 & C1 & C2 \\ C3 & C4 & C5 \\ C6 & C7 & C8 \end{pmatrix} \times \begin{pmatrix} R_i \\ G_i \\ B_i \end{pmatrix}$$

For calculation fixed point precision values are used. Parameters *Precision* and *Resolution* specify the multiplication precisions.

- *Precision* defines the number of bits used to encode the coefficients.
- *Resolution* defines the number of fractional bits of a coefficient.

For example a precision of 8 bit and a resolution of 6 bits will result in coefficients within the range $\left[-2^{Precision-Resolution-1} \text{ to } 2^{Precision-Resolution-1} - \frac{1}{2^{Resolution}}\right] = \left[-2 \text{ to } 2 - \frac{1}{64}\right]$, $\text{Stepsize } \frac{1}{2^{Resolution}} = \frac{1}{64}$.

Signed values are allowed as coefficients, if the input link is signed or the field parameter *Coefficients* is set to static. When the field parameter *Coefficients* is set to dynamic (i.e., transformation coefficients can be altered during runtime), the output link is always signed. One of the precision bits is used as the signed bit.

The results of the matrix multiplication are provided. They are rounded to the next integer value. The output bit width is automatically determined from the input bit width and parameter settings.



**Less Resources for Static Parameter**

The operator requires less FPGA resources if the parameter *Coefficients* is set to static.

## 22.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data input |

## 22.3.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [3, 54] unsigned, [6, 54] signed | auto |
| **Arithmetic** | {unsigned, signed} | auto |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | as I |
| **Color Flavor** | FL_RGB | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The output bit width is automatically determined from the input bit width and the parameter settings.

### 22.3.3. Parameters

| **Resolution** | |
|---|---|
| **Type** | static parameter |
| **Default** | 8 |
| **Range** | auto |

Defines the number of fractional bits of a coefficient. The range depends on the input bit width and the value of parameter Precision. Mind the following limitations:

Resolution > 0

Precision > Resolution

Precision + Resolution <= 32

$(InputBitWidth/3 + Precision - Resolution + 2) \times 3 < 64$

| **Precision** | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | auto |

Defines the number of bits to encode the whole coefficient - resolution bits and sign bit included. The range depends on the input bit width and the value of parameter Precision. Mind the following limitations:

Resolution > 0

Precision > Resolution

Precision + Resolution <= 32

$(InputBitWidth/3 + Precision - Resolution + 2) \times 3 < 64$

| **Coefficients** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | identity matrix |
| **Range** | All Coefficients unsigned: $\left[0 \text{ to } 2^{Precision-Resolution} - \frac{1}{2^{Resolution}}\right]$, Stepsize $\frac{1}{2^{Resolution}}$ Any<br>Coefficient signed: $\left[-2^{Precision-Resolution-1} 0 \text{ to } 2^{Precision-Resolution-1} - \frac{1}{2^{Resolution}}\right]$, Stepsize $\frac{1}{2^{Resolution}}$ |

The coefficient matrix initialized with the identity transformation. All coefficients of the matrix are treated as signed numbers by the operator. The coefficients are double numbers.



**Only Embedded VisualApplets (eVA): Deviating Parameter Interface during Runtime**

During runtime on eVA platforms, this parameter shows another parameter interface than during design time in the VisualApplets GUI: Field parameter *Coefficents* is replaced by separate index parameters and value parameters (parameter

| Coefficients | |
|---|---|
| | names: *CoefficientIndex* and *CoefficientValue*). The value is written on access to *CoefficientValue*. |

### 22.3.4. Examples of Use

The use of operator ColorTransform is shown in the following examples:

- Section 13.8, 'Functional Example for Specific Operators of Library Color, Base and Memory'

  Examples - Demonstration of how to use the operator

# 22.4. Operator HSI2RGB

**Operator Library: Color**

The operator HSI2RGB converts the color space from HSI (Hue Saturation Lightness) to RGB (Red, Green, Blue).



**Warning**

The operator implements the HSL (Hue Saturation Lightness) to RGB color space conversion and not HSI to RGB.

**Operator Features and Restrictions**

- Empty Images (i.e., images with no pixels) are fully supported.
- Not empty Images with empty lines are fully supported.
- Images with varying line lengths are fully supported.

## 22.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data input |

## 22.4.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [24, 48]❶ | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | as I |
| **Color Flavor** | FL_HSI | FL_RGB |
| **Max. Img Width** | any❷ | as I |
| **Max. Img Height** | any | as I |

❶ The Bit Width must be divisible by three without remainder.
❷ VALT_IMAGE2D and VALT_LINE1D: The Maximal Image Width must be divisible by the parallelism without remainder (not important for VALT_PIXEL0D and VALT_SIGNAL).

## 22.4.3. Parameters

| HueAmplitude | |
|---|---|
| **Type** | static parameter |
| **Default** | 255 |
| **Range** | [1; (1 << (BitWidth(I) / 3)) - 1] |

| HueAmplitude |
| --- |
| This parameter defines the maximum value for the hue. 9bit at 360 degrees corresponds to a result in degrees. |



**Range Violation leads to DRC error**

If the value of HueAmplitude is not in range, the parameter state is set to error and the parameter is reported by the DRC.

### 22.4.4. Examples of Use

The use of operator HSI2RGB is shown in the following examples:

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

# 22.5. Operator RGB2HSI

**Operator Library: Color**

The operator RGB2HSI converts the color space from RGB (Red, Green, Blue) to HSL (Hue Saturation Luminance).



**Warning**

The operator implements the RGB to HSL (Hue Saturation Luminance) color space conversion and not RGB to HSI (Hue Saturation Intensity).

**Operator Features and Restrictions**

- Empty Images (i.e., images with no pixels) are fully supported.
- Not empty Images with empty lines are fully supported.
- Images with varying line lengths are fully supported.

## 22.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data input |

## 22.5.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [24, 48]❶ | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | as I |
| **Color Flavor** | FL_RGB | FL_HSI |
| **Max. Img Width** | any❷ | as I |
| **Max. Img Height** | any | as I |

❶ The Bit Width must be divisible by three without remainder.
❷ VALT_IMAGE2D and VALT_LINE1D: The Maximal Image Width must be divisible by the parallelism without remainder (not important for VALT_PIXEL0D and VALT_SIGNAL).

## 22.5.3. Parameters

| HueAmplitude | |
|---|---|
| **Type** | static parameter |
| **Default** | 255 |
| **Range** | [1; (1 << (BitWidth(I) / 3)) - 1] |

| HueAmplitude |
| --- |
| This parameter defines the maximum value for the hue. 9bit at 360 degrees corresponds to a result in degrees. |

**Range Violation leads to DRC error**

If the value of HueAmplitude is not in range, the parameter state is set to error and the parameter is reported by the DRC.

### 22.5.4. Examples of Use

The use of operator RGB2HSI is shown in the following examples:

- Section 12.4.3, 'HSL Color Classification'

  Examples - Color Classification is very simple on HSL images. The applet converts the RGB image into an HSL image and performs a color classification. The hue is filtered using a lookup table. Moreover, the saturation and lightness is thresholded using custom threshold values.

# 22.6. Operator RGB2YUV

**Operator Library: Color**

The module RGB2YUV converts the color space from RGB to YCbCr.



**Operator converts RGB into YCbCr, not into YUV**

The operator converts only RGB into YCbCr. It does not convert RGB into YUV.

**Operator Features and Restrictions**

- Empty Images (i.e., images with no pixels) are fully supported.
- Not empty Images with empty lines are fully supported.
- Images with varying line lengths are fully supported.

## 22.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data input |

## 22.6.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 24 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | as I |
| **Color Flavor** | FL_RGB | FL_YUV |
| **Max. Img Width** | any❶ | as I |
| **Max. Img Height** | any | as I |

❶ VALT_IMAGE2D and VALT_LINE1D: The Maximal Image Width must be divisible by the parallelism without remainder (not important for VALT_PIXEL0D and VALT_SIGNAL).

## 22.6.3. Parameters

None

## 22.6.4. Examples of Use

The use of operator RGB2YUV is shown in the following examples:

- Section 13.8, 'Functional Example for Specific Operators of Library Color, Base and Memory'

Examples - Demonstration of how to use the operator

# 22.7. Operator WhiteBalance

**Operator Library: Color**

The operator WhiteBalance scales the three color components of an RGB input stream using three independent coefficients. One usage of the operator is manual white balancing. Each color component is scaled with the values defined by parameter *RedCoefficient*, *GreenCoefficient* and *BlueCoefficient*. The range and precision of the multiplication is defines using parameters *ResolutionBits* and *RangeBits*. *RangeBits* defines the scaling range. The range is [0, 2^RangeBits[. *ResolutionBits* defines the step size of the scaling which is $\frac{1}{2^{ResolutionBits}}$ . The results are rounded to the next integer values and output on output link O.

## 22.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data input |

## 22.7.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [3, 63] | 3*(InputBitWidth/3 + RangeBits) <= 64 |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | as I |
| **Color Flavor** | FL_RGB | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 22.7.3. Parameters

| **ResolutionBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 8 |
| **Range** | [1, 16] |
| This parameter defines the precision of the multiplication i.e. the step size of the coefficient parameters. | |

| **RangeBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 2 |
| **Range** | [1, 8] |
| This parameter defines the range of the multiplication. This parameter influences the output bit width. The output bit width must not exceed 64 Bit. | |

| RedCoefficient | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | $\left[0 \text{ to } 2^{RangeBits} - \frac{1}{2^{ResolutionBits}}\right]$, Stepsize $\frac{1}{2^{ResolutionBits}}$ |
| This parameter defines the coefficient which scales the red component of the input. The entered floating point coefficient is rounded to the nearest valid fixed point value (see RangeBits and ResolutionBits). | |

| GreenCoefficient | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | $\left[0 \text{ to } 2^{RangeBits} - \frac{1}{2^{ResolutionBits}}\right]$, Stepsize $\frac{1}{2^{ResolutionBits}}$ |
| This parameter defines the coefficient which scales the green component of the input. The entered floating point coefficient is rounded to the nearest valid fixed point value (see RangeBits and ResolutionBits). | |

| BlueCoefficient | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | $\left[0 \text{ to } 2^{RangeBits} - \frac{1}{2^{ResolutionBits}}\right]$, Stepsize $\frac{1}{2^{ResolutionBits}}$ |
| This parameter defines the coefficient which scales the blue component of the input. The entered floating point coefficient is rounded to the nearest valid fixed point value (see RangeBits and ResolutionBits). | |

## 22.7.4. Examples of Use

The use of operator WhiteBalance is shown in the following examples:

- Section 12.4.4, 'RGB White Balancing'

  Examples - The applet shows an example for white balancing on RGB images.

# 22.8. Operator WhiteBalanceBayer

**Operator Library: Color**

The operator WhiteBalanceBayer scales the three color components of a Bayer input stream using three independent coefficients. One usage of the operator is manual white balancing. Each color component is scaled with the values defined by parameter *RedCoefficient*, *GreenCoefficient* and *BlueCoefficient*. The range and precision of the multiplication is defines using parameters *ResolutionBits* and *RangeBits*. *RangeBits* defines the scaling range. The range is [0, 2^RangeBits[. *ResolutionBits* defines the step size of the scaling which is $\frac{1}{2^{ResolutionBits}}$ . The results are rounded to the next integer values and output on output link O.

The Bayer array configuration of the camera must match the operator settings. The operator setting is changed using parameter *BayerInit*.

## 22.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data input |

## 22.8.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 16] | InputBitWidth + RangeBits |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 22.8.3. Parameters

| BayerInit | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | GreenFollowedByRed |
| **Range** | {GreenFollowedByRed, GreenFollowedByBlue, RedFollowedByGreen, BlueFollowedByGreen} |
| The Bayer array configuration of the camera must match the operator settings. | |

| ResolutionBits | |
|---|---|
| **Type** | static parameter |
| **Default** | 8 |
| **Range** | [1, 16] |

| ResolutionBits |
|---|
| This parameter defines the precision of the multiplication i.e. the step size of the coefficient parameters. |

| RangeBits | |
|---|---|
| **Type** | static parameter |
| **Default** | 2 |
| **Range** | [1, 8] |
| This parameter defines the range of the multiplication. This parameter influences the output bit width. The output bit width must not exceed 64 Bit. | |

| RedCoefficient | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | $\left[0 \text{ to } 2^{RangeBits} - \frac{1}{2^{ResolutionBits}}\right]$, Stepsize $\frac{1}{2^{ResolutionBits}}$ |
| This parameter defines the coefficient which scales the red component of the input. The entered floating point coefficient is rounded to the nearest valid fixed point value (see RangeBits and ResolutionBits). | |

| GreenCoefficient | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | $\left[0 \text{ to } 2^{RangeBits} - \frac{1}{2^{ResolutionBits}}\right]$, Stepsize $\frac{1}{2^{ResolutionBits}}$ |
| This parameter defines the coefficient which scales the green component of the input. The entered floating point coefficient is rounded to the nearest valid fixed point value (see RangeBits and ResolutionBits). | |

| BlueCoefficient | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | $\left[0 \text{ to } 2^{RangeBits} - \frac{1}{2^{ResolutionBits}}\right]$, Stepsize $\frac{1}{2^{ResolutionBits}}$ |
| This parameter defines the coefficient which scales the blue component of the input. The entered floating point coefficient is rounded to the nearest valid fixed point value (see RangeBits and ResolutionBits). | |

## 22.8.4. Examples of Use

The use of operator WhiteBalanceBayer is shown in the following examples:

- Section 12.4.1.4, 'Bayer 3x3 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Moreover, a white balancing for color correction is added.

# 23. Library Compression



The *Compression* library includes operators for image compressions such as JPEG.

**Availability**

To use the **Compression** library, you need either a **JPEG Compression Library** license, or the **VisualApplets 4** license.

The following list summarizes all Operators of Library Compression

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | ImageBuffer_JPEG_Gray | Buffers the image stream in on-board RAM and remaps the pixels in 8x8 blocks for JPEG encoding. | Version 1.3 |
|  | JPEG_Encoder_Gray | Performs a JPEG compression of the previously remapped input stream. | Version 1.3 |
|  | JPEG_Encoder | Performs a JPEG compression of the previously remapped input stream. | Version 1.3 |

Table 23.1. Operators of Library Compression

# 23.1. Operator ImageBuffer_JPEG_Gray

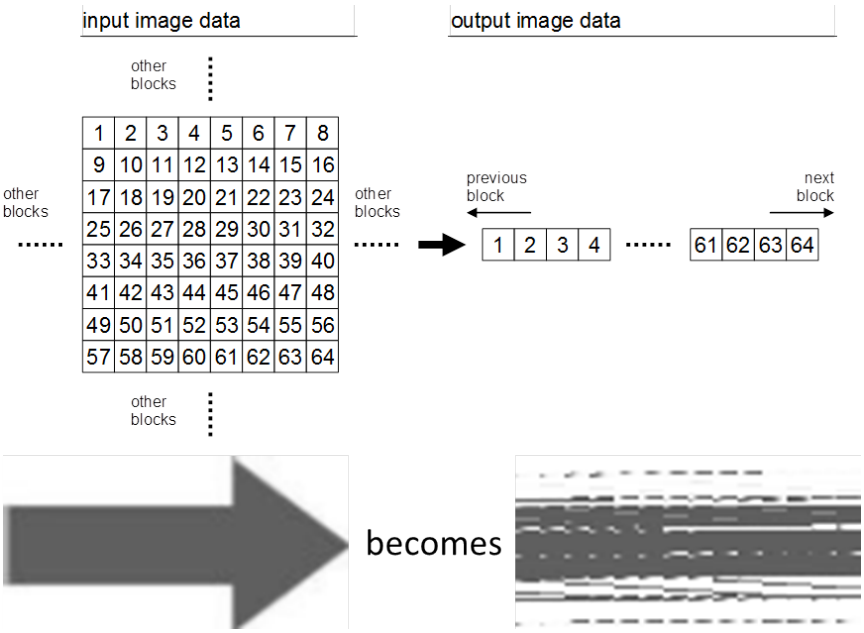
**Operator Library: Compression**

The operator ImageBuffer_JPEG_Gray is used in combination with operator JPEG_Encode_Gray. The operator buffers the image data and remap pixels for the successive JPEG encoder.

**Availability**

To use the *ImageBuffer_JPEG_Gray* operator, you need either a **JPEG Compression Library** license, or the **VisualApplets 4** license.

For JPEG compression images have to be divided into blocks of eight by eight pixels. The JPEG encoder requires an input data stream consisting of these previously remapped pixels which is performed by the ImageBuffer_JPEG_Gray. The following figure illustrates the behavior




Besides the sorting of pixels and the buffering of image data, the operator allows a dynamic ROI selection. The XLength has to be a multiple of 16 where the XOffset has to be a multiple of eight. In y-direction the offset may be any value, where the YLength has to be a multiple of 8. The operator always cuts the given XLength. If the input image is smaller than the parameterized XLength, the current memory content will be output. If the y length of the input image is smaller than the y length given by the parameters, the output of the buffer is clipped the same y length as the input plus an extension so that the height is a multiple of eight.

The ROI coordinates may be changed at any time. However, a change while the operator is currently processing an image might result in unwanted results. Therefore, a change during idle periods is recommended.

**Operator Restrictions**

- The operator does not support empty images i.e. images with no pixels.
- The lines of each input image at port I must have the same length. Thus images with varying line lengths are not allowed.
- The operator requires non FPGA memory. The memory resources might be limited. Check the available memory resources. (mostly DRAM memory) Map available RAM resources to modules of this operators. See Section 3.8, 'Allocation of Device Resources' for more information.

| Availability for Hardware Platforms |
|---|
| Please note that this operator is only available for target platforms of the microEnable 4 series (including PixelPlant). |

### 23.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 23.1.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 8 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | {1, 2, 4, 8}❶ | 8 |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any❷ | as I |
| **Max. Img Height** | any | as I |

❶ The maximum parallelism depends on the available RAM data width on the frame grabber. Parallelism eight is only available for RAM data widths greater equal 64.

The RAM data width of all microEnable IV frame grabbers is 64.

❷ The maximum image width is

$$2^{RamAddressWidth}/4$$

The RAM address width of all microEnable IV frame grabbers is 24 bit.

### 23.1.3. Parameters

| XOffset | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, MaxImageWidth - 16], step size = 8 |
| This parameter defines the x-coordinate of the upper left corner of the ROI. The XOffset has to be a multiple of eight.<br>$XOffset + XLenght \leq OutputMaxImageWidth$ | |

| XLength | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 1024 |
| **Range** | [16, MaxImageWidth], step size = 16 |

| XLength |
|---|
| This parameter defines the width of the ROI. It has to be a multiple of 16.<br><br>$XOffset + XLenght \leq OutputMaxImageWidth$ |

| YOffset | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, MaxImageHeight - 8], step size = 1 |
| This parameter defines the y-coordinate of the upper left corner of the ROI. The YOffset has to be a multiple of eight.<br><br>$YOffset + YLenght \leq OutputMaxImageHeight$ | |

| YLength | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 1024 |
| **Range** | [8, MaxImageHeight], step size = 8 |
| This parameter defines the height of the ROI. It has to be a multiple of 8.<br><br>$YOffset + YLenght \leq OutputMaxImageHeight$ | |

| FillLevel | |
|---|---|
| **Type** | static read parameter |
| **Default** | 0% |
| **Range** | [0, 100%] |
| This read only parameter shows the current fill level of the buffer. | |

| InfiniteSource | |
|---|---|
| **Type** | static parameter |
| **Default** | ENABLED |
| **Range** | {ENABLED, DISABLED} |
| Set this parameter to ENABLED if the operator is connected to a camera without any other buffers in between. Otherwise set the value to DISABLED. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

### 23.1.4. Examples of Use

The use of operator ImageBuffer_JPEG_Gray is shown in the following examples:

- Section 12.1.2, 'JPEG Encoder Gray'

  Examples - A simple example which shows the usage of the JPEG operators.

## 23.2. Operator JPEG_Encoder_Gray

**Operator Library: Compression**

The operator performs a JPEG compression of grayscale 8-bit images.



**Availability**

To use the *JPEG_Encoder_Gray* operator, you need either a **JPEG Compression Library** license, or the **VisualApplets 4** license.



**Input Data in 8x8 pixels blocks required**

On its input, the operator needs to receive image data that have been remapped into blocks of 8x8 pixels.

If you use a microEnable IV frame grabber, you can use operator *ImageBuffer_JPEG_Gray* for this remapping. For microEnable IV designs, simply use operator *ImageBuffer_JPEG_Gray* in front of operator *JPEG_Encoder_Gray*. Operator *ImageBuffer_JPEG_Gray* is designed to perform this reordering.

If you use a microEnable 5 frame grabber, make sure you remap the image data into blocks of 8x8 pixels (using the general VisualApplet options) before you use operator *JPEG_Encoder_Gray*.

The operator's output is a Huffman stream without JPEG headers. The compression rate depends on the selected quantization table which is changeable during runtime. For Huffman coding the standard luminance table is used. Besides the quantization table, the compression rate i.e. the output stream size depends on the input image. Therefore, the operator's output width is changeable to the desired output size.

The JPEG encoder is able to process around 2.5 input pixels per clock cycle. The clock frequency depends on the used frame grabber. microEnable IV frame grabbers have a base clock frequency of 62.5MHz. Therefore, the operator can process 158MPixels/s on a microEnable IV frame grabber. If quantization tables that cause a high quality compression are used, the data rate can be reduced. This is at quality levels of approximately more than 95%. Please refer to the JPEG application notes for further details.

The operator allows the selection of the quantization table using two parameters representing two different ways of configuration.

- If desired, each of the quantization table values can be set individually. Use parameter *quantization_matrix* in this case.
- In alternative, the quantization table can be determined from percentage values. Use parameter *quality_in_percent* in this case.

The standard luminance quantization table is shown in the following and is used as the default setting of the operator.

$$Q_{orig}(u,v) = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}$$

If a percentage quality is used, the quantization table is calculated by the following equation

$$Q(u,v) = \begin{cases} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix} & \text{if } q = 100 \\ Q(u,v) = \text{round}\left(Q_{orig}(u,v)\frac{100-q}{50}\right) & \text{if } q \geq 50 \\ Q(u,v) = \text{round}\left(Q_{orig}(u,v)\frac{50}{q}\right) & \text{if } q \leq 50 \end{cases}$$

As described, the operator's output is a Huffman Stream of the encoded image data. The underlying DC coefficients and AC coefficients are fixed and taken from literature, namely W. P. Pennbaker and J. L. Mitchell, JPEG Still Image Data Compression Standard, Van Nostrand Rheinhold, 1993. See beyond for the listing. The generated Huffman Stream generated by the encoder does not contain the required JPEG headers to generate a JPEG image file. Furthermore, the generated data stream is of 32Bbit width. The last word i.e. the four last byte of the JPEG data stream provide status information of the following format:

- Bytes zero and one of the last word represent the JPEG EOI marker which is 0xff and 0xd9.
- Byte no. two represents the number of bytes used in the second last data word in the range [1, 4] i.e. the true end of the JPEG data stream.
- The last byte no. three represents an error code. If the JPEG data stream is larger than the maximum image width specified for the output link, the output will be truncated. If this happens, the last byte will be set to one which allows the detection of this truncation.

An example of the end of a JPEG output data stream is given in the following:

Second last word: 0xffff5fed

Last word: 0x0002d9ff

Hence, the JPEG output data stream ends with the second byte of the second last word. The data stream is not truncated.

**Operator Restrictions**

- The operator does not support empty images i.e. images with no pixels.
- The lines of each input image at port I must have the same length. Thus images with varying line lengths are not allowed.

## 23.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 23.2.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 8 | 32 |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 8 | 1 |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | VAF_GRAY | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any❶ | any |
| **Max. Img Height** | any❷ | 1 |

❶❷ The input width and height have to be multiples of 8.

### 23.2.3. Parameters

| quality_in_percent | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 50 |
| **Range** | [-1, 100] |
| Using this parameter the quality of the JPEG compression can be changed. The quantization matrix is determined from the percentage values using the equation given above. The determined quantization values can be read from parameter quantization_matrix. The parameter is dynamic and should only be changed during idle time of the applet.<br><br>Quality settings between 0 and 100 can be defined. Writing to this parameter overwrites manual chnages of the quantization matrix made by parameter *quantization_matrix*. If -1 is read from *quality_in_percent*, manual change of the quantization matrix has been made. | |

| quantization_matrix | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | standard matrix (see above) |
| **Range** | [1, 255] |
| This parameter is a dynamic read and write matrix paramter. If the JPEG quality is set in percentage values. This parameter can be used to read the determined quantization matrix values.<br><br>The parameter can also be used to specify a user defined quantization matrix. This can be done by writing to any of the quantization matrix values. Note that a write to parameter qulity_in_percent will discard the user specified quantization values. For convenience, parameter *quality_in_percent* is set to -1 after a user specified value has been written. | |

### 23.2.4. Examples of Use

The use of operator JPEG_Encoder_Gray is shown in the following examples:

- Section 12.1.2, 'JPEG Encoder Gray'

  Examples - A simple example which shows the usage of the JPEG operators.

- Section 12.1.3, 'Using more than one JPEG encoder to enhance the bandwidth of JPEG compression.'

  Examples - multible JPEG encoder

### 23.2.5. More Information

The following Huffman DC- and AC Coefficients are used.

```
typedef char DCHuffTableType[12][17]; // huffman-table for luminance-DC-coefficients

DCHuffTableType Lum_DC_HuffmanTable= {
"00",
"010",
"011",
"100",
"101",
```

```
"110",
"1110",
"11110",
"111110",
"1111110",
"11111110",
"111111110" };

typedef char ACHuffTableType[16][11][17]; // Huffman-Tabelle für Luminance-AC-Koeffizienten

ACHuffTableType Lum_AC_HuffmanTable= {
{ //Run == 0
"1010",//EOB
"00",
"01",
"100",
"1011",
"11010",
"1111000",
"11111000",
"1111110110",
"1111111110000010",
"1111111110000011"
},
{ //Run == 1
"1010",//EOB
"1100",
"11011",
"1111001",
"111110110",
"11111110110",
"1111111110000100",
"1111111110000101",
"1111111110000110",
"1111111110000111",
"1111111110001000"
},
{ //Run == 2
"1010",//EOB
"11100",
"11111001",
"1111110111",
"111111110100",
"1111111110001001",
"1111111110001010",
"1111111110001011",
"1111111110001100",
"1111111110001101",
"1111111110001110"
},
{ //Run == 3
"1010",//EOB
"111010",
"111110111",
"111111110101",
"1111111110001111",
"1111111110010000",
"1111111110010001",
"1111111110010010",
"1111111110010011",
"1111111110010100",
"1111111110010101"
},
{ //Run == 4
"1010",//EOB
"111011",
"1111111000",
"1111111110010110",
"1111111110010111",
"1111111110011000",
"1111111110011001",
"1111111110011010",
"1111111110011011",
"1111111110011100",
"1111111110011101",
},
{ //Run == 5
"1010",//EOB
"1111010",
"11111110111",
```

```
"1111111110011110",
"1111111110011111",
"1111111110100000",
"1111111110100001",
"1111111110100010",
"1111111110100011",
"1111111110100100",
"1111111110100101"
},
{ //Run == 6
"1010",//EOB
"1111011",
"111111110110",
"1111111110100110",
"1111111110100111",
"1111111110101000",
"1111111110101001",
"1111111110101010",
"1111111110101011",
"1111111110101100",
"1111111110101101"
},
{ //Run == 7
"1010",//EOB
"11111010",
"111111110111",
"1111111110101110",
"1111111110101111",
"1111111110110000",
"1111111110110001",
"1111111110110010",
"1111111110110011",
"1111111110110100",
"1111111110110101",
},
{ //Run == 8
"1010",//EOB
"111111000",
"111111111000000",
"1111111110110110",
"1111111110110111",
"1111111110111000",
"1111111110111001",
"1111111110111010",
"1111111110111011",
"1111111110111100",
"1111111110111101"
},
{ //Run == 9
"1010",//EOB
"111111001",
"1111111110111110",
"1111111110111111",
"1111111111000000",
"1111111111000001",
"1111111111000010",
"1111111111000011",
"1111111111000100",
"1111111111000101",
"1111111111000110"
},
{ //Run == 0xA
"1010",//EOB
"111111010",
"1111111111000111",
"1111111111001000",
"1111111111001001",
"1111111111001010",
"1111111111001011",
"1111111111001100",
"1111111111001101",
"1111111111001110",
"1111111111001111"
},
{ //Run == 0xB
"1010",//EOB
"1111111001",
"1111111111010000",
"1111111111010001",
"1111111111010010",
```

```
"1111111111010011",
"1111111111010100",
"1111111111010101",
"1111111111010110",
"1111111111010111",
"1111111111011000"
},
{ //Run == 0xC
"1010",//EOB
"1111111010",
"1111111111011001",
"1111111111011010",
"1111111111011011",
"1111111111011100",
"1111111111011101",
"1111111111011110",
"1111111111011111",
"1111111111100000",
"1111111111100001"
},
{ //Run == 0xD
"1010",//EOB
"11111111000",
"1111111111100010",
"1111111111100011",
"1111111111100100",
"1111111111100101",
"1111111111100110",
"1111111111100111",
"1111111111101000",
"1111111111101001",
"1111111111101010"
},
{ //Run == 0xE
"1010",//EOB
"1111111111101011",
"1111111111101100",
"1111111111101101",
"1111111111101110",
"1111111111101111",
"1111111111110000",
"1111111111110001",
"1111111111110010",
"1111111111110011",
"1111111111110100"
},
{ //Run == 0xF
"11111111001", //ZRL
"1111111111110101",
"1111111111110110",
"1111111111110111",
"1111111111111000",
"1111111111111001",
"1111111111111010",
"1111111111111011",
"1111111111111100",
"1111111111111101",
"1111111111111110"
}
};
```

## 23.3. Operator JPEG_Encoder

**Operator Library: Compression**

The operator performs a JPEG compression of grayscale 8-bit images. It uses the JPEG baseline algorithm. The operator's output is a Huffman stream. Optionally, JPEG headers are included in the output (parametrizable). If the headers are included, the output format is JFIF (JPEG File Interchange Format), version 1.2.



**Availability**

To use the *JPEG_Encoder* operator, you need either a **JPEG Compression Library** license, or the **VisualApplets 4** license.

The compression rate (and thus the output stream size) depends on two factors:

- on the selected quantization table which is changeable during runtime.
- on the input image.

Operator *JPEG_Encoder* is able to process the full input data rate as specified in the link parametrization. You can define the throughput rate via the input parallelism. Please note that higher parallelism entails a higher FPGA resource utilization.

The maximum image height is 65.535 pixels. If the image height is not a multiple of eight, the operator internally adds dummy lines. This behaviour reduces the overall input data rate.

The operator allows you to define the quantization table. You have two options to configure the table:

- The quantization table can be calculated automatically on the basis of a quality value (in percent). For calculation, the standard luminance quantization table (see below) is used. Use parameter *Quality* (in percent) to configure automatic calculation. Automatic calculation is the default setting of the operator.
- Alternatively, you can set each of the quantization table values individually. Use parameter *LuminanceQuantization* to enter your values. Parameter *Quality* is automatically disabled in this case by being set to value -1.

The standard luminance quantization table (default setting) looks as follows:

$$Q_{orig}(u,v) = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}$$

In default mode (i.e. automatic calculation out of the standard lumninance quantization table set by the parameter *Quality*), the quantization table is calculated by the following equation:

$$Q(u,v) = \begin{cases} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix} & \text{if } q = 100 \\ Q(u,v) = \text{round}\left(Q_{orig}(u,v)\dfrac{100-q}{50}\right) & \text{if } q \geq 50 \\ Q(u,v) = \text{round}\left(Q_{orig}(u,v)\dfrac{50}{q}\right) & \text{if } q \leq 50 \end{cases}$$

As described above, the operator's output is a Huffman stream of the encoded image data. For Huffman coding, the standard luminance tables for DC and AC coefficients are used. These tables are fixed and taken from literature, namely W. P. Pennbaker and J. L. Mitchell, 'JPEG Still Image Data Compression Standard', Van Nostrand Rheinhold, 1993. The Huffman stream generated by the encoder includes

the JPEG header if parameter *IncludeHeader* is set to YES. The generated data stream consists of parallel outputs of multiple bytes (8-bit blocks) where the parallelism is automatically derived from the throughput requirements.

**Operator Restrictions**

- The operator does not support empty images, i.e., images with no pixels.
- Input images with varying line lengths are not allowed.
- The operator has a minimum input image width which depends on the input parallelism. The minimum input image width is at least twice the input parallelism. The minimum input image width is calculated as follows:

  1. On the basis of the input parallelism: The first multiple of 8 is identified which is equal to or bigger than the input parallelism.
  2. To this multiple of 8, value 1 is added.
  3. On the basis of the result of step 2: As the input image width has to be a multiple of the input parallelism, the first multiple of the **input parallelism** that is bigger than the result of step 2 is identified.

  The result of step 3 is the minimum input image width for the operator.

$$MinWidth = ceil\left(\frac{ceil\left(\frac{par}{8}\right)\cdot 8+1}{par}\right)\cdot par$$

Figure 23.1. Formula for calculating the minimum input image width

For example, if the input parallelism is 4, the minimum input image width is 12.

If the input parallelism is 12, the minimum input image width is 24.



**Optimizing the Operator Throughput**

If the input parallelism is greater than 8, the data throughput depends on the size of the input image. The maximum data throughput can be achieved with the following image size (OptimalSize):

**OptimalSize = Ceil(FrameSize/(PathCount * IntervalSize))*PathCount * IntervalSize**

FrameSize = ceil(ImageWidth/8)*8 * ceil(ImageHeight/8)*8;

PathCount = ceil(Par/8)

IntervalSize: Next value greater than 7, which has no common factor with PathCount.

Internally, the operator always uses a parallelism that is a multiple of 8. Depending on the input parallelism, the internal parallelism conversion can compensate some of the loss of bandwidth. The loss of bandwidth only occurs at the end of a frame.

As the size of the compressed image data is not predictable, the last output data byte of a compressed frame might not occur aligned to the output parallelism. Consequently, the last data vector of a frame can contain random dummy values. To mark the actual end of a frame, the EOI (End of Image) marker as defined in the JPEG standard will be used. The EOI marker consists of two bytes. The second last byte of each frame is 0xFF, and the last byte of each frame is 0xD9.

To optimize its image throughput rate (band width), the operator outputs the header as soon as header generation is activated - even before image data arrive at the operator's input. This way, the transfer of the header data doesn't interrupt the transfer of image data, as the header is transferred in advance.

The drawback of this practice is that the operator's output transfer starts earlier than the actual image data transfer. This may cause irritations under specific circumstances:

- Using operator SourceSelector directly after JPEG_Encoder: Operator SourceSelector registers a partly processed frame as soon as it gets the header data. Therefore, if SourceSelector is switched to getting image data from JPEG_Encoder, SourceSelector cannot be switched to any another source as it always detects an unfinished frame. In addition, when header generation is enabled and SourceSelector switches from another source to the JPEG_Encoder channel, the first image is lost.
- To measure the latency, use operator FrameEndToSignal (instead of FrameStartToSignal and SignalToDelay).
- If working on eVA devices, please make sure the output is capable to accept data transfer before the sensor transfer is started.

Get more information and additional examples on VisualApplets' JPEG compression features in the VisualAppletsOnline Community Forum [https://forum.silicon.software/forum/] at New High-Speed JPEG Operator and Examples [https://forum.silicon.software/forum/index.php?thread/100-new-high-speed-jpeg-operator-and-examples-va-3-2-0/].

**Overflow Management with InfiniteSource**

In the *InfiniteSource* mode it is possible for images to be lost or corrupted, because the *JPEG_Encoder* module or one of the succeeding modules can't handle the bandwidth. If the overflow occurs while a partial image was already accepted by the operator, all further incoming image data is discarded and the truncated image in the operator is truncated, which results in a partial output image. If the operator is in an overflow state while the start of a frame arrives, the entire frame is discarded and the frame is lost. For each truncated or lost frame, a VA event is generated (`TruncatedEvent` and `LostEvent`). The *JPEG_Encoder* events consist of 3 packets with 2 byte each. The first two VA event packets make up the frame-ID, which is just a counter for each frame that arrives at the *JPEG_Encoder* input. The frame-ID identifies the exact frame that was lost or truncated. The third packet marks the type of error that occurred. Bit 0 marks whether a frame was lost (Bit0 = 1) or truncated (Bit0 = 0). The other 15 bit of the third packet are only used in case events were lost, which can only occur if events occur too fast for the CPU. Bit1 of the third packet marks any occurrence of a `TruncatedEvent` that was lost and Bit2 of the same packet marks any occurrence of a `LostEvent` that was lost. Additionally, for `LostEvents` that were lost, Bit[15:3] form a counter that holds the number of `LostEvents` that were lost. If the counter value is zero, the lost-`LostEvent`-counter has overflown. Since a truncated frame results in *JPEG_Encoder* output data but a lost frame does not, the VA events only count the number of lost *LostEvents*.

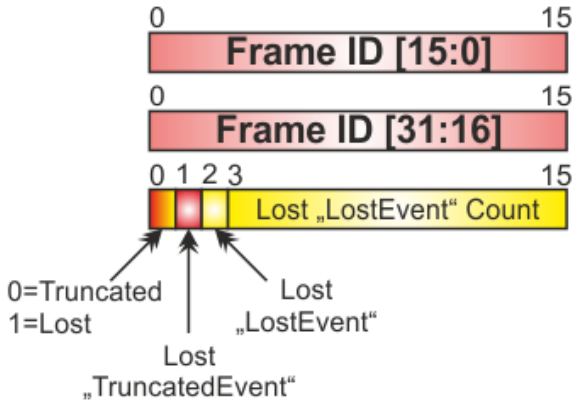



Figure 23.2. Overflow Event Data

## 23.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |

| Property | Value |
|---|---|
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 23.3.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 8 | 8 |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | automatically calculated |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | 2^16 -1 = 65.535❶ | any❷ |
| **Max. Img Height** | 2^16 -1 = 65.535 | 1 |

❶ The value must not be lower than the minimum image width requirement.
❷ The image width at the output link is configurable. If the output image is bigger than the maximal image width configured for the output link, the image is cut off and the remaining image data is discarded. The last two bytes of each image are always containing the End of Image Marker (EOI).

### 23.3.3. Parameters

| Quality | |
|---|---|
| **Type** | static/dynamic write parameter |
| **Default** | 50.00 |
| **Range** | [1.00 - 100.00 %], {-1}, step size = 0.01% |

Using this parameter, the quality of the JPEG compression can be changed. The quantization matrix is determined from the percentage values using the equation given above. The determined quantization values can be read from parameter quantization_matrix. The parameter is dynamic and should only be changed during idle time of the applet.

Quality settings between 1 and 100 can be defined. Writing to this parameter overwrites manual changes of the quantization matrix made by parameter *LuminanceQuantization*. If -1 is read from *Quality*, manual change of the quantization matrix has been made.

The quality parameter is the primary source to define the compression rate of the encoder. The luminance table will be auto-computed from the specified quality and can be read back. However it is possible to modify the luminance table directly. If this happens the quality parameter will be auto-changed to -1 to show that the parameter is not valid anymore and manual overwrite mode for the tables is used. The quality and the quantization tables can be set to static mode if the user wants to optimize resource usage. Static versus dynamic change can be performed only on the quality parameter. The quantization table settings will follow the quality type automatically and cannot be overwritten manually. When the quality is set to static, the operator determines the output parallelism from the quality settings. In most cases the output parallelism will be reduced in comparison to the dynamic mode, which can mean a significant FPGA resource reduction at the cost of giving up the flexibility to change the compression rate during the runtime.

| LuminanceQuantization | |
|---|---|
| **Type** | follows Quality type (dynamic or static) write parameter |
| **Default** | none |

| LuminanceQuantization | |
|---|---|
| **Range** | [1 - 255] |
| auto computed for quality in range [1.00 - 100.00], when manually set, quality is invalidated to -1. | |

| IncludeHeader | |
|---|---|
| **Type** | static write parameter |
| **Default** | YES |
| **Range** | [YES,NO] |
| The JPEG header is per default included in the compression stream. However, you can disable this feature. If you set parameter *IncludeHeader* to NO, the JPEG parameters ImageHeight and ImageWidth will become deactivated. If you set parameter *IncludeHeader* to YES, the header is included and you can decide if the header parameters can be static or are required to be dynamic. If set to static, the values for image width and image height will be statically embedded into the header and cannot be changed, regardless of the input image size. If set to dynamic, you can change the values for image width and image height during the runtime. Please note that these header parameters are not automatically updated to the input image size. If you use the operator with images the size of which is dynamically changing during runtime, you will have to patch the produced header afterwards. If you set ImageHeight and ImageWidth both to "static", you will achieve a slight reduction of the FPGA resource usage. | |

| ImageWidth | |
|---|---|
| **Type** | static/dynamic write parameter |
| **Default** | 1024 |
| **Range** | 1 - 2^16-1 |
| This parameter is only available if parameter *IncludeHeader* is set to YES.<br><br>Parameter *ImageWidth* is only used for generating the JPEG image header as described in parameter *IncludeHeader*. | |

| ImageHeight | |
|---|---|
| **Type** | static/dynamic write parameter |
| **Default** | 1024 |
| **Range** | 1 - 2^16-1 |
| This parameter is only available if parameter *IncludeHeader* is set to YES.<br><br>Parameter *ImageHeight* is only used for generating the JPEG image header as described in parameter *IncludeHeader*. | |

| InfiniteSource | |
|---|---|
| **Type** | static write parameter |
| **Default** | DISABLED |
| **Range** | {ENABLED, DISABLED} |
| The operator can be plugged directly after a camera operator. In this case the *InfiniteSource* parameter must be set to ENABLED. Then the operator will perform active overflow management and report overflow conditions to the software through VA event system. The overflow can occur only in 2 situations: the sink behind the operator will stop/pause the transmission or the input image height is not a multiple of 8 lines. In the latter case the operator has to pad the missing lines to complete the last row JPEG blocks as required in JPEG standard.<br><br>See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

## 23.3.4. Examples of Use

The use of operator JPEG_Encoder is shown in the following examples:

- Section 12.1.4, 'JPEG Compression Using Operator **JPEG_Encoder**'

  Examples - Simple examples which show the usage of the operator **JPEG_Encoder**.

## 23.3.5. More Information

The following Huffman DC and AC coefficients are used.

```
typedef char DCHuffTableType[12][17]; // Huffman table for
    luminance DC coefficients DCHuffTableType Lum_DC_HuffmanTable= { "00", "010", "011", "100",
    "101", "110", "1110", "11110", "111110", "1111110", "11111110", "111111110" }; typedef char
    ACHuffTableType[16][11][17]; // Huffman table for luminance AC coefficients ACHuffTableType
    Lum_AC_HuffmanTable= { { //Run == 0 "1010",//EOB "00", "01", "100", "1011", "11010",
    "1111000", "11111000", "1111110110", "1111111110000010", "1111111110000011" }, { //Run == 1
    "1010",//EOB "1100", "11011", "1111001", "111110110", "11111110110", "1111111110000100",
    "1111111110000101", "1111111110000110", "1111111110000111", "1111111110001000" }, { //Run == 2
    "1010",//EOB "11100", "11111001", "1111110111", "111111110100", "1111111110001001",
    "1111111110001010", "1111111110001011", "1111111110001100", "1111111110001101",
    "1111111110001110" }, { //Run == 3 "1010",//EOB "111010", "111110111", "111111110101",
    "1111111110001111", "1111111110010000", "1111111110010001", "1111111110010010",
    "1111111110010011", "1111111110010100", "1111111110010101" }, { //Run == 4 "1010",//EOB
    "111011", "1111111000", "1111111110010110", "1111111110010111", "1111111110011000",
    "1111111110011001", "1111111110011010", "1111111110011011", "1111111110011100",
    "1111111110011101", }, { //Run == 5 "1010",//EOB "1111010", "11111110111", "1111111110011110",
    "1111111110011111", "1111111110100000", "1111111110100001", "1111111110100010",
    "1111111110100011", "1111111110100100", "1111111110100101" }, { //Run == 6 "1010",//EOB
    "1111011", "111111110110", "1111111110100110", "1111111110100111", "1111111110101000",
    "1111111110101001", "1111111110101010", "1111111110101011", "1111111110101100",
    "1111111110101101" }, { //Run == 7 "1010",//EOB "11111010", "111111110111",
    "1111111110101110", "1111111110101111", "1111111110110000", "1111111110110001",
    "1111111110110010", "1111111110110011", "1111111110110100", "1111111110110101", }, { //Run ==
    8 "1010",//EOB "111111000", "111111111000000", "1111111110110110", "1111111110110111",
    "1111111110111000", "1111111110111001", "1111111110111010", "1111111110111011",
    "1111111110111100", "1111111110111101" }, { //Run == 9 "1010",//EOB "111111001",
    "1111111110111110", "1111111110111111", "1111111111000000", "1111111111000001",
    "1111111111000010", "1111111111000011", "1111111111000100", "1111111111000101",
    "1111111111000110" }, { //Run == 0xA "1010",//EOB "111111010", "1111111111000111",
    "1111111111001000", "1111111111001001", "1111111111001010", "1111111111001011",
    "1111111111001100", "1111111111001101", "1111111111001110", "1111111111001111" }, { //Run ==
    0xB "1010",//EOB "1111111001", "1111111111010000", "1111111111010001", "1111111111010010",
    "1111111111010011", "1111111111010100", "1111111111010101", "1111111111010110",
    "1111111111010111", "1111111111011000" }, { //Run == 0xC "1010",//EOB "1111111010",
    "1111111111011001", "1111111111011010", "1111111111011011", "1111111111011100",
    "1111111111011101", "1111111111011110", "1111111111011111", "1111111111100000",
    "1111111111100001" }, { //Run == 0xD "1010",//EOB "11111111000", "1111111111100010",
    "1111111111100011", "1111111111100100", "1111111111100101", "1111111111100110",
    "1111111111100111", "1111111111101000", "1111111111101001", "1111111111101010" }, { //Run ==
    0xE "1010",//EOB "1111111111101011", "1111111111101100", "1111111111101101",
    "1111111111101110", "1111111111101111", "1111111111110000", "1111111111110001",
    "1111111111110010", "1111111111110011", "1111111111110100" }, { //Run == 0xF "11111111001",
    //ZRL "1111111111110101", "1111111111110110", "1111111111110111", "1111111111111000",
    "1111111111111001", "1111111111111010", "1111111111111011", "1111111111111100",
    "1111111111111101", "1111111111111110" } };
```

Further operators ave available as user library elements, see Section 4.2.8, 'Delivered User Libraries' for details.

# 24. Library Debugging




**Runtime Analysis:** The *Debugging* library allows you to analyze VisualApplets designs (that do not work as expected) *during runtime*.

The data provided by the individual operators will help you to

- debug your **VisualApplet Design** and to improve its stability.
- debug the **Custom Operators** you are developing: You can use the operators of the debugging library to analyze the effects of your custom operators in designs.



**Runtime Testing**

The debugging library is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before the operators provide data you can use for debugging.

The operators of the debugging libary have nothing to do with design simulation within VisualApplets.

The Debugging Library offers dedicated operators you can use to

- Analyze the data stream in an image processing pipeline:
  - Image statistics (number of frames, image width/image height, line frequency, image frequency, detection of varying line length and especially of empty lines)
  - Data flow analysis (blocking statistics, bandwidth)
- Manipulate the image data stream by:
  - Intentionally blocking the data stream
  - Suppressing the natural blocking of the image stream
- Test individual parts of your VA designs via:
  - Image emulators that you can configure to show most diverse timing behaviour
- Insert and monitor image data at any place within a design:
  - Image injection via register I/O
  - Image monitoring via register I/O
- Analyze signals



**Availability**

To use the **Debugging** library, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

The following list summarizes all Operators of Library Debugging

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | ImageAnalyzer | Performs an analysis of image properties without touching the image data. | Version 3.0.1 |
|  | ImageStatistics | Tracks the most important parameters of an image stream. Calculates average line and frame frequencies. | Version 3.0.1 |
|  | StreamAnalyzer | Performs an analysis of the current flow control properties (blocked/not blocked) of the data stream without touching the image data. | Version 3.0.1 |
|  | Scope | Provides means for drawing a waveform from sampled input data. | Version 3.0.1 |
|  | ImageInjector | Allows to inject image data into the processing pipeline by register I/O. | Version 3.0.1 |
|  | ImageTimingGenerator | Creates a black image output with adjustable timing setting. | Version 3.0.1 |
|  | ImageFlowControl | Allows to handle overflow situations. | Version 3.0.1 |
|  | StreamControl | Allows to suppress the blocking status as well as to create blocking pulses towards the input link. | Version 3.0.1 |
|  | ImageMonitor | Allows probing image data from the processing pipeline by register I/O. | Version 3.0.1 |

Table 24.1. Operators of Library Debugging

# 24.1. Operator ImageAnalyzer

**Operator Library: Debugging**

Operator *ImageAnalyzer* analyzes image properties without touching the image data.

The operator analyzes individual images. For analyzing image sequences, use operator *ImageStatistics*.



**Availability**

To use the *ImageAnalyzer* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

Operator *ImageAnalyzer* provides

- data about the last frame (values for the complete frame), and
- intermediate values for the current frame (the frame that is currently streamed).

Operator *ImageAnalyzer* analyzes frames in area scan applications (2D). You can also use it for analyzing data in line scan applications (1D).

The operator offers dynamic read parameters you can use to retrieve data about

- image dimension (width x height) and image size (pixel),
- the deviation of line lengths within a frame,
- time gaps between lines,
- the blocking state of the operator input (blocked/not blocked).

In addition, the operator offers some parameters to control the operator itself.

When an internal counter overflows, a corresponding overflow bit is set in the *OverflowMask* parameter and the counter halts.



**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

The operator is not intended for design simulation within VisualApplets.

## 24.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 24.1.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 24.1.3. Parameters

| **ClearMode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | ClearWithProcessStart |
| **Range** | {NoClearing, ClearWithProcessStart, ClearWithProcessReset, SendClearPulse} |

**Operator Control:**

This parameter defines the reset behavior for all **read** parameters.

**NoClearing**: Values of all *read* parameters are held. ProcessEnable and ProcessReset have no influcence on these values.

**ClearWithProcessStart**: Values of all *read* parameters are only cleared with rising edge of ProcessEnable.

**ClearWithProcessReset**: Values of all *read* parameters are only cleared with rising edge of ProcessReset.

**SendClearPulse**: On-demand clearing of all *read* parameter values.

| **FrameCountWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 24 |
| **Range** | [4, 63] |

**Operator Control:**

Sets the bit width of parameter *FrameCounter*.

| **GapCountWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 20 |
| **Range** | [4, 63] |

**Operator Control:**

Sets the bit width of the line gap counters (*FrameMinLineGap, FrameMaxLineGap, CurrLineGap*).

| **FrameCount** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^FrameCountWidth) -1] |

**General Information:**

Counts all images which are terminated with an EndOfFrame.

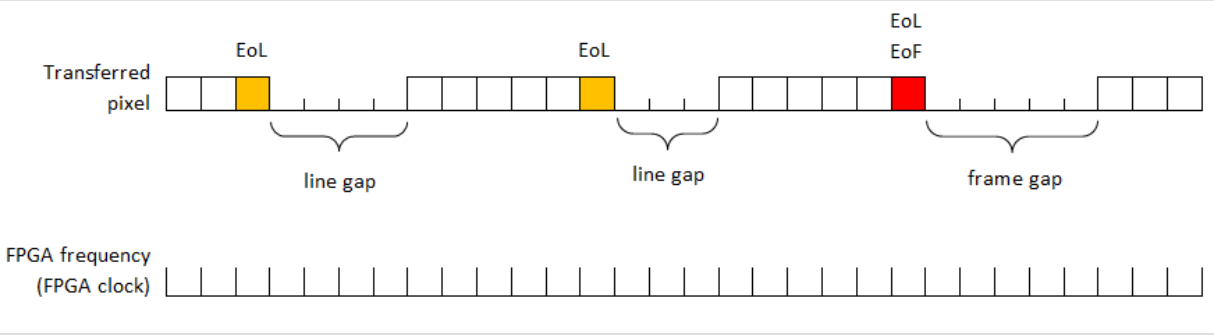
| FramePixelCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^ (ImageWidthBitwidth+ImageHeightBitwidth+4))-1] |
| **Last Frame:** | |
| Size of the last completed frame in pixels. | |

| FrameHeight | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageHeightBitwidth+4))-1] |
| **Last Frame:** | |
| Height of the last completed frame in pixels. | |

| FrameMinWidth | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |
| **Last Frame:** | |
| Minimal width encountered in last completed frame (in pixels). | |

| FrameMaxWidth | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |
| **Last Frame:** | |
| Maximal width encountered in last completed frame (in pixels). | |

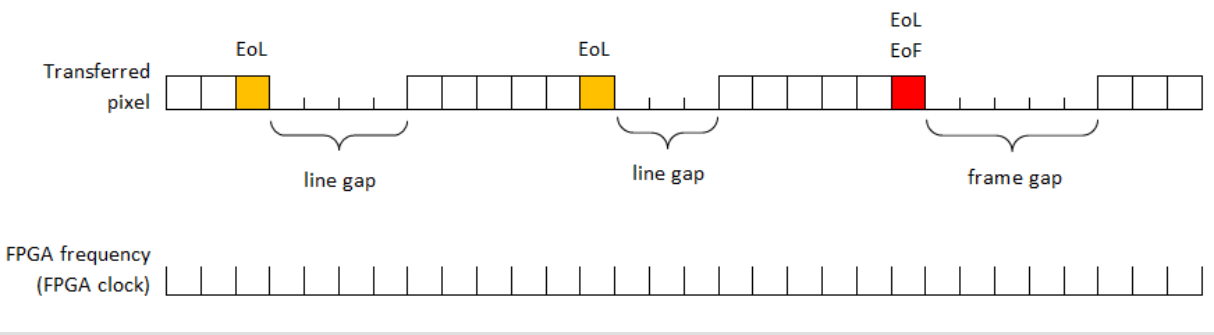
| FrameMinLineGap | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^GapCountWidth)-1] |
| **Last Frame:** | |
| Minimal line gap encountered in last completed frame. In the example displayed in the figure below, FrameMinLineGap = 3. | |



| FrameMaxLineGap | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+ImageHeightBitwidth+4))-1] |

**Last Frame:**

Maximal line gap encountered in last completed frame. In the example displayed in the figure below, FrameMaxLineGap = 4.




| CurrPixelCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+ImageHeightBitwidth+4))-1] |

**Current Frame:**

Pixel count in the frame that is currently active and under inspection.

| CurrXPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |

**Current Frame:**

Current horizontal position in the frame that is currently active and under inspection.

| CurrYPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageHeightBitwidth+4))-1] |

**Current Frame:**

Current vertical position in the frame that is currently active and under inspection.

| CurrMinWidth | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |

**Current Frame:**

Minimal width detected in the frame that is currently active and under inspection.

| CurrMaxWidth | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |

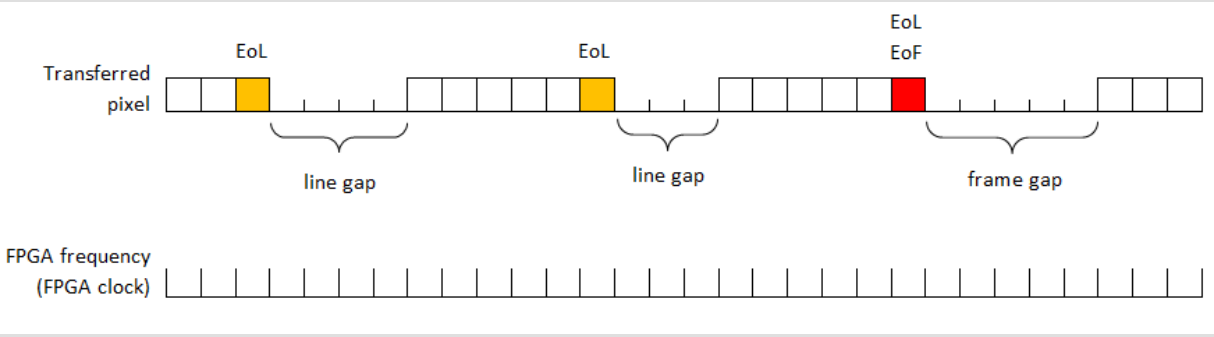
**Current Frame:**

Maximal width detected in the frame that is currently active and under inspection.

| CurrLineGap | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^GapCountWidth)-1] |

**Current Frame:**

Gap between current line and preceding line (in frame that is currently active and under inspection).




| CurrBlocked | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |

**General Information:**

If set to one: Value 1 informs that the input link is currently blocked by the output link.

| OverflowMask | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 31] |

**Operator Control:**

Bit-encoded overflow signaling. A set bit indicates an overflow in the following counters:

[0] = FrameCount

[1] = PixelCount

[2] = xPos

[3] = yPos

[4] = LineGapCount.

### 24.1.4. Examples of Use

The use of operator ImageAnalyzer is shown in the following examples:

- Section 12.6.2, 'Image Dimension Test'

  Example - The image dimension is measured and can be used to analyze the design flow.

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

# 24.2. Operator ImageStatistics

**Operator Library: Debugging**

Operator *ImageStatistics* measures frame properties for whole frame sequences without touching the image data.



**Availability**

To use the *ImageStatistics* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

Operator *ImageStatistics* analyzes image sequences. For analyzing individual images, use operator Operator *ImageAnalyzer*.

The operator allows to analyze deviations between frames (width, height, size, line gaps, etc.). The operator also provides measurands regarding line rates and frame rates.

In addition, operator *ImageStatistics* provides data about the utilization of the pipeline capacity (valid fraction, idle fraction, blocked fraction of time in percent).

When an internal counter overflows, a corresponding overflow bit is set in the OverflowMask parameter and the counter halts.



**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

The operator is not intended for design simulation within VisualApplets.

## 24.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data output |

## 24.2.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 24.2.3. Parameters

| **ClearMode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | ClearWithProcessStart |
| **Range** | {NoClearing, ClearWithProcessStart, ClearWithProcessReset, SendClearPulse} |
| This parameter defines the reset behavior for all **read** parameters.<br><br>**NoClearing**: Values of all read parameters are held. ProcessEnable and ProcessReset have no influcence on these values.<br><br>**ClearWithProcessStart**: Values of read parameters are only cleared with rising edge of ProcessEnable.<br><br>**ClearWithProcessReset**: Values of read parameters are only cleared with rising edge of ProcessReset.<br><br>**SendClearPulse**: On-demand clearing of all read parameter values. | |

| **MeasurementPeriod** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 1000 |
| **Range** | [1, 65535] |
| Time in milliseconds for rate and duty cycle measurements. . | |

| **FrameCountWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 24 |
| **Range** | [4, 63] |
| Sets the bit width of the frame counter (parameter *FrameCount*). | |

| **GapCountWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 20 |
| **Range** | [4, 63] |
| Sets the bit width of the line gap counters (MinLineGap, MaxLineGap, MeanLineGap, MinLinePeriod, MaxLinePeriod, MeanLinePeriod). | |

| **FrameCount** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^FrameCountWidth) -1] |
| Counts all images which are terminated with an EndOfFrame. Available only in IMAGE-2D mode. | |

| **FramePixelCount** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^ (ImageWidthBitwidth+ImageHeightBitwidth+4))-1] |
| Size of the last completed frame in pixels. | |

| **MinWidth** | |
|---|---|
| **Type** | dynamic read parameter |

| MinWidth | |
|---|---|
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |
| Minimal line width detected in all frames inspected so far (in pixels). | |

| MaxWidth | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |
| Maximal line width detected in all frames inspected so far (in pixels). | |

| MinHeight | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageHeightBitwidth+4))-1] |
| Minimal image height detected in all frames inspected so far (in pixels). | |

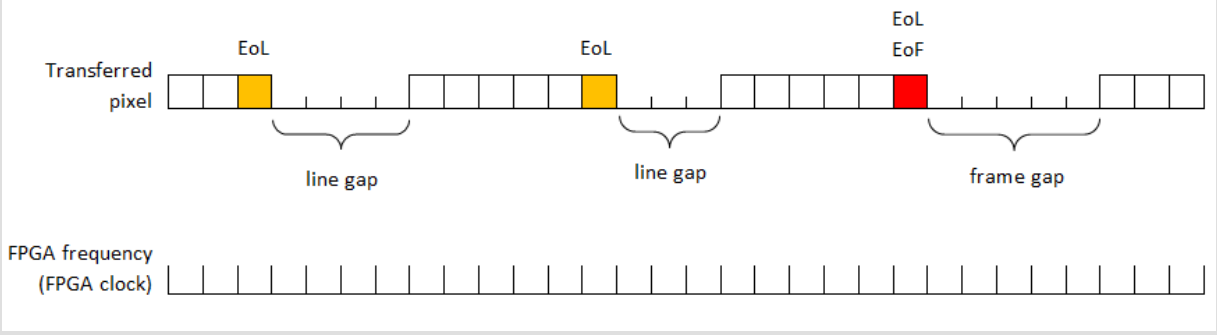
| MaxHeight | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageHeightBitwidth+4))-1] |
| Maximal image height detected in all frames inspected so far (in pixels). | |

| MinPixelCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+ImageHeightBitwidth+4))-1] |
| Size of smallest frame out of all frames inspected so far (in pixels). | |

| MaxPixelCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+ImageHeightBitwidth+4))-1] |
| Size of biggest frame out of all frames inspected so far (in pixels). | |

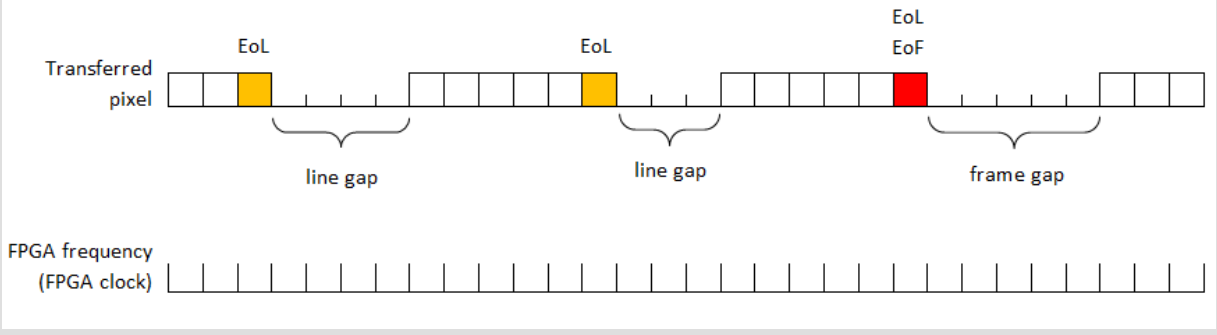
| MinLineGap | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^GapCountWidth)-1] |
| Minimal line gap encountered in all frames inspected so far. Does not include the frame gap between the last line of a 2D image and the first line of the next image. | |

| MinLineGap |
|---|



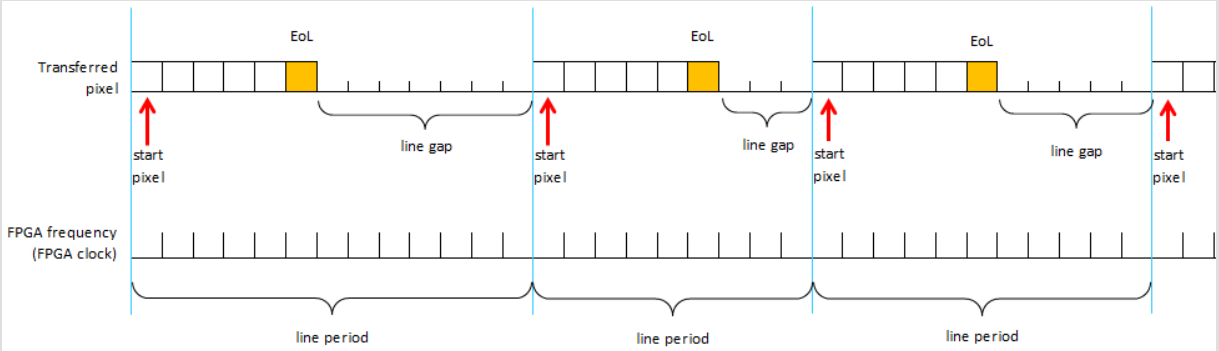

| MaxLineGap | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^GapCountWidth)-1] |

Maximal line gap encountered encountered in all frames inspected so far. Does not include the frame gap between the last line of a 2D image and the first line of the next image.




| MinLinePeriod | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^GapCountWidth)-1] |

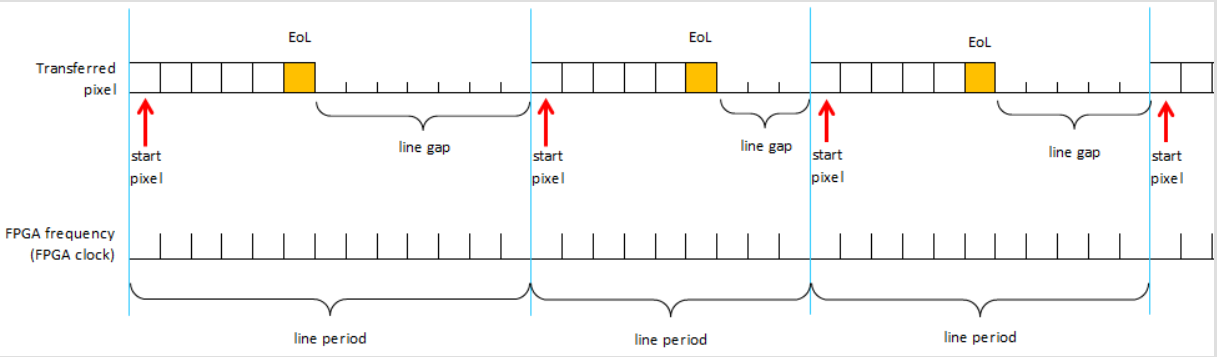
Minimal line period encountered. A Line period is the number of clock cycles between two line beginnings. A line period thus covers the line width, a single EndOfLine pulse, and the line gap. The frame gap between the last line of a 2D image and the first line of the next image is not included.




| MaxLinePeriod | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |

| MaxLinePeriod | |
|---|---|
| **Range** | [0, (2^GapCountWidth)-1] |

Maximal line period encountered. A Line period is the number of clock cycles between two line beginnings. A line period thus covers the line width, a single EndOfLine pulse, and the line gap. The frame gap between the last line of a 2D image and the first line of the next image is not included.

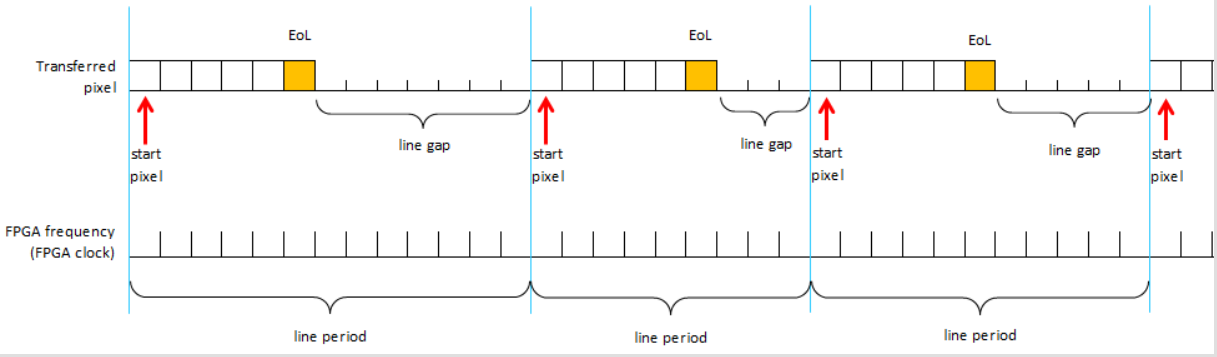



| MeanWidth | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |

Mean line width during set MeasurementPeriod (in pixels).

| MeanLinePeriod | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^GapCountWidth)-1] |

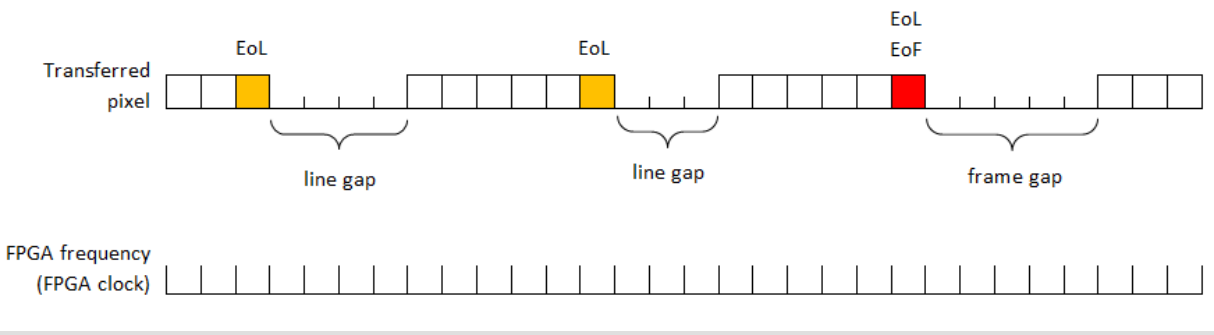
Mean line period during set MeasurementPeriod. A Line period is the number of clock cycles between two line beginnings. A line period thus covers the line width, a single EndOfLine pulse, and the line gap. The frame gap between the last line of a 2D image and the first line of the next image is not included.

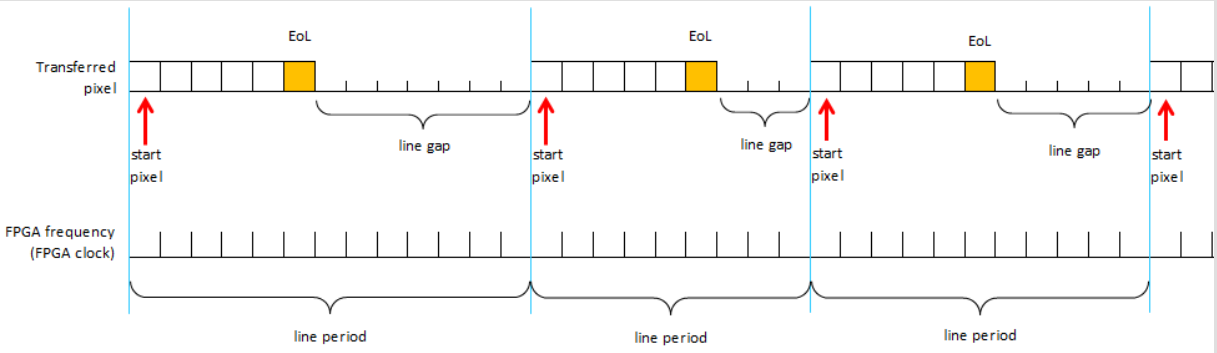



| MeanLineGap | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^GapCountWidth)-1] |

Mean line gap during set MeasurementPeriod (mean number of clock cycles between two line beginnings).

The frame gap between the last line of a 2D image and the first line of the next image is not included.

| MeanLineGap |
|---|
|  |

| MeanLineDutyCycle | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0.0, 100.0] |
| Mean line duty cycle during set MeasurementPeriod. Defined as MeanWidth divided by MeanLinePeriod. | |
|  | |

| LineRate | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | |
| Line rate in Hertz. | |

| FrameRate | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | |
| Frame rate in Hertz. | |

| BlockedFraction | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0.0, 100.0] |
| Percentage of blocked cycles. | |

| IdleFraction | |
|---|---|
| **Type** | dynamic read parameter |

| IdleFraction | |
|---|---|
| **Default** | 0 |
| **Range** | [0.0, 100.0] |
| Percentage of idle cycles. | |

| ValidFraction | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0.0, 100.0] |
| Percentage of valid cycles carrying data. | |

| MaxWidthErrorCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitwidth+4))-1] |
| Counts up if line width is greater than maximum line width defined on the input link. | |

| MaxHeightErrorCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^( ImageHeightBitwidth+4))-1] |
| Counts up if image height is greater than maximum image height defined on the input link. | |

| OverflowMask | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 4095] |

Bit encoded overflow signaling. A set bit indicates an overflow in the following counters:

[0] = FrameCount

[1] = PixelCount

[2] = xPos

[3] = yPos

[4] = LineGap

[6] = MeanLineGap

[8] = BlockedFraction

[10] = ValidFraction

[11] = LineCounter. Invalidates LineRate, MeanWidth, MeanLinePeriod and MeanLineGap.

### 24.2.4. Examples of Use

The use of operator ImageStatistics is shown in the following examples:

- Section 12.6.7, 'Image Flow Control'

  Example - For debugging purposes of the designs internal data flow control in hardware and a possible compensation.

# 24.3. Operator StreamAnalyzer

**Operator Library: Debugging**

Operator *StreamAnaylzer* provides information about the data flow through. It doesn't touch the image data.



**Availability**

To use the *StreamAnaylzer* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

**You can use the operator to detect blocking (inhibit) conditions.**

When a blocking occurs in the pipeline, you can use this operator to see where in a frame (at which pixel) the blocking occured.

In addition, operator *StreamAnalyzer* provides data about the utilization of the pipeline capacity (valid fraction, idle fraction, blocked fraction of time in percent).

When an internal counter overflows, a corresponding overflow bit is set in the *OverflowMask* parameter and the counter halts.



**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

The operator is not intended for design simulation within VisualApplets.

## 24.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, Image data input |
| **Output Link** | O, Image data output |

## 24.3.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | VALT_IMAGE2D, LINE1D, PIXEL0D | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 24.3.3. Parameters

| ClearMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | ClearWithProcessStart |
| **Range** | {NoClearing, ClearWithProcessStart, ClearWithProcessReset, SendClearPulse} |

This parameter defines the reset behavior for all **read** parameters.

**NoClearing**: Values of all read parameters are held. ProcessEnable and ProcessReset have no influcence on these values.

**ClearWithProcessStart**: Values of read parameters are only cleared with rising edge of ProcessEnable.

**ClearWithProcessReset**: Values of read parameters are only cleared with rising edge of ProcessReset.

**SendClearPulse**: On-demand clearing of all read parameter values.

| MeasurementMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | AllCycles |
| **Range** | {AllCycles, FrameCyclesOnly, LineCyclesOnly} |

This parameter allows you to analyze specific fractions of the MeasurementPeriod:

**AllCycles**: Performance is measured over all cycles inside the MeasurementPeriod.

**FrameCyclesOnly**: Performance is measured over all cycles between the start end the end of frames. Frame gaps inside the MeasurementPeriod are therefore omitted.

**LineCyclesOnly**: Performance is measured over all cycles between the start end the end of lines. Frame gaps and line gaps inside the MeasurementPeriod are therefore omitted.

| MeasurementPeriod | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1000 |
| **Range** | [1, 65535] |

Time in milliseconds for performance measurements.

The definition of the measurement period is important for analyzing time fractions, and for calculating mean values.

| FrameCountWidth | |
|---|---|
| **Type** | static write parameter |
| **Default** | 24 |
| **Range** | [4, 63] |

Sets the bit width of the frame counter. (Frame counter is parameter*CurrFrame*).

| CurrTime | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^32)-1] |

Time passed since the beginning of the measurement period (in milliseconds).

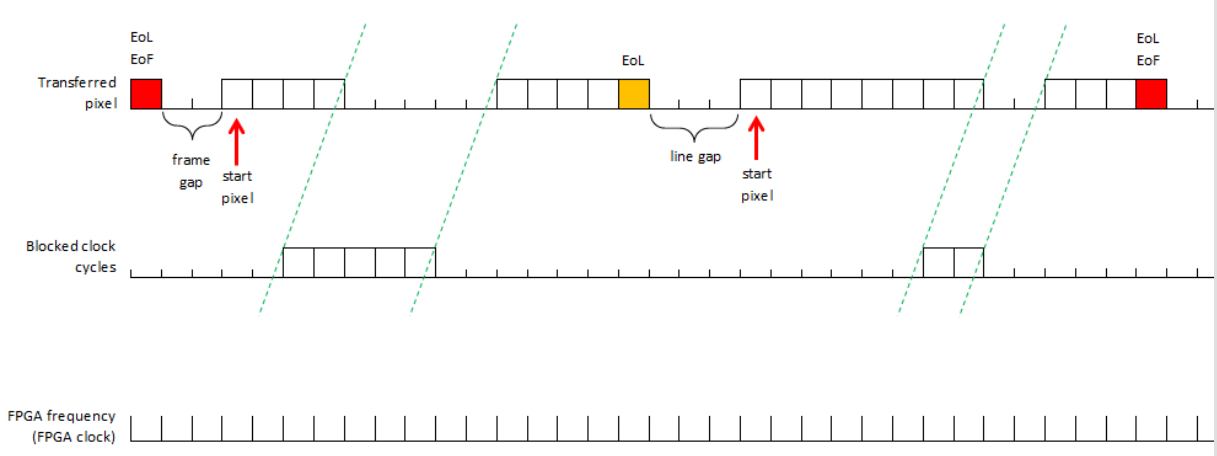
| CurrFrame | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^FrameCountWidth) -1] |
| This parameter acts as a frame counter. Counted are all frames which are terminated with an EndOfFrame flag. Available only in IMAGE-2D mode. | |

| CurrBlocked | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| If set to one: Value 1 informs that the image flow is blocked. | |

| BlockedCyclesPerFrame | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthtBitwidth+4))-1] |

Counts the number of clock cycles during which the image flow is blocked during the flow-through of one frame.

A pixel gap always has exactly the length of the blocking which provokes the gap. However, the offset of the gap may differ from the offset of the blocking status.




If buffer operators have been implemented earlier in the processing line, the pixels that can not be transferred due to the blocking status are buffered.

If no buffer operator is implemented earlier in the processing line, the pixels that can not be transferred due to the blocking status are discarded.

| FirstBlockedFrame | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^FrameCountWidth)-1] |
| Frame number of frame during which first blocking occurred. | |

| LastBlockedFrame | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |

| LastBlockedFrame | |
|---|---|
| **Range** | [0, (2^FrameCountWidth)-1] |
| Frame number of frame during which last blocking has occurred. | |

| FirstBlockedXPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitWidth+4))-1] |
| Horizontal position at which first blocking appeared. | |

| LastBlockedXPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitWidth+4))-1] |
| The last horizontal position at which the image flow was blocked until now.<br><br>Example: If the value is 7: Horizontal position 7 was the last X position blocked until now. Horizontal position 8 was the first unblocked position after the last blocking that occurred. | |

| FirstBlockedYPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitWidth+4))-1] |
| Vertical position at which first blocking appeared. | |

| LastBlockedYPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^(ImageWidthBitWidth+4))-1] |
| The last vertical position at which the image flow was blocked until now.<br><br>Example: If the value is 13: Vertical position 13 was the last Y position blocked until now. Vertical position 14 was the first unblocked position after the last blocking that occurred. | |

| FirstBlockedTime | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^32)-1] |
| Time stamp at which first blocking appeared. | |

| LastBlockedTime | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, (2^32)-1] |
| Time stamp of point of time at which last blocking has appeared.<br><br>After this point of time, no further blocking occured until now. | |

| DataRate | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |

| DataRate | |
|---|---|
| **Range** | |
| Measured data rate in mega pixel per second. | |

| BlockedFraction | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0.0, 100.0] |
| Percentage of blocked cycles within the measurement period (see parameter MeasurementPeriod). | |

| IdleFraction | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0.0, 100.0] |
| Percentage of idle cycles within the measurement period (see parameter MeasurementPeriod). | |

| ValidFraction | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0.0, 100.0] |
| Percentage of valid cycles carrying data within the measurement period (see parameter MeasurementPeriod). | |

| OverflowMask | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 511] |

**Operator Control:**

Bit-encoded overflow signaling. A set bit indicates an overflow in the following counters:

[0] = FrameCount

[1] = PixelCount

[2] = xPos

[3] = yPos

[4] = LinePeriod

[5] = InhibitCount

[6] = InhibitCountFrame

[7] = IdleCount

[8] = ValidCount

### 24.3.4. Examples of Use

The use of operator StreamAnalyzer is shown in the following examples:

- Section 12.6.2, 'Image Dimension Test'

Example - The image dimension is measured and can be used to analyze the design flow.

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

## 24.4. Operator Scope

**Operator Library: Debugging**

The Scope operator provides options for analyzing gray-scale pictures.



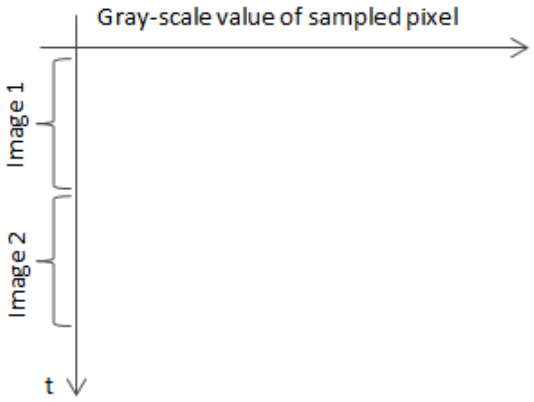
**Availability**

To use the *Scope* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

The operator outputs a 2D waveform for each image channel by sampling input image lines. Up to four channels are supported.

In a channel, each incoming line is sampled once. According to the settings you define in parameter SampleMode, the gray scale value of the first pixel in the line, the last pixel in the line, the smallest pixel value in the line, or the greatest pixel value in the line is used for sampling.

The sampled pixel value is used to set an output pixel in a co-ordinate system. The horizontal position of the output pixel is defined by the actual value of the sampled pixel. The vertical position corresponds to the order of incoming lines, i.e., to the time axis.

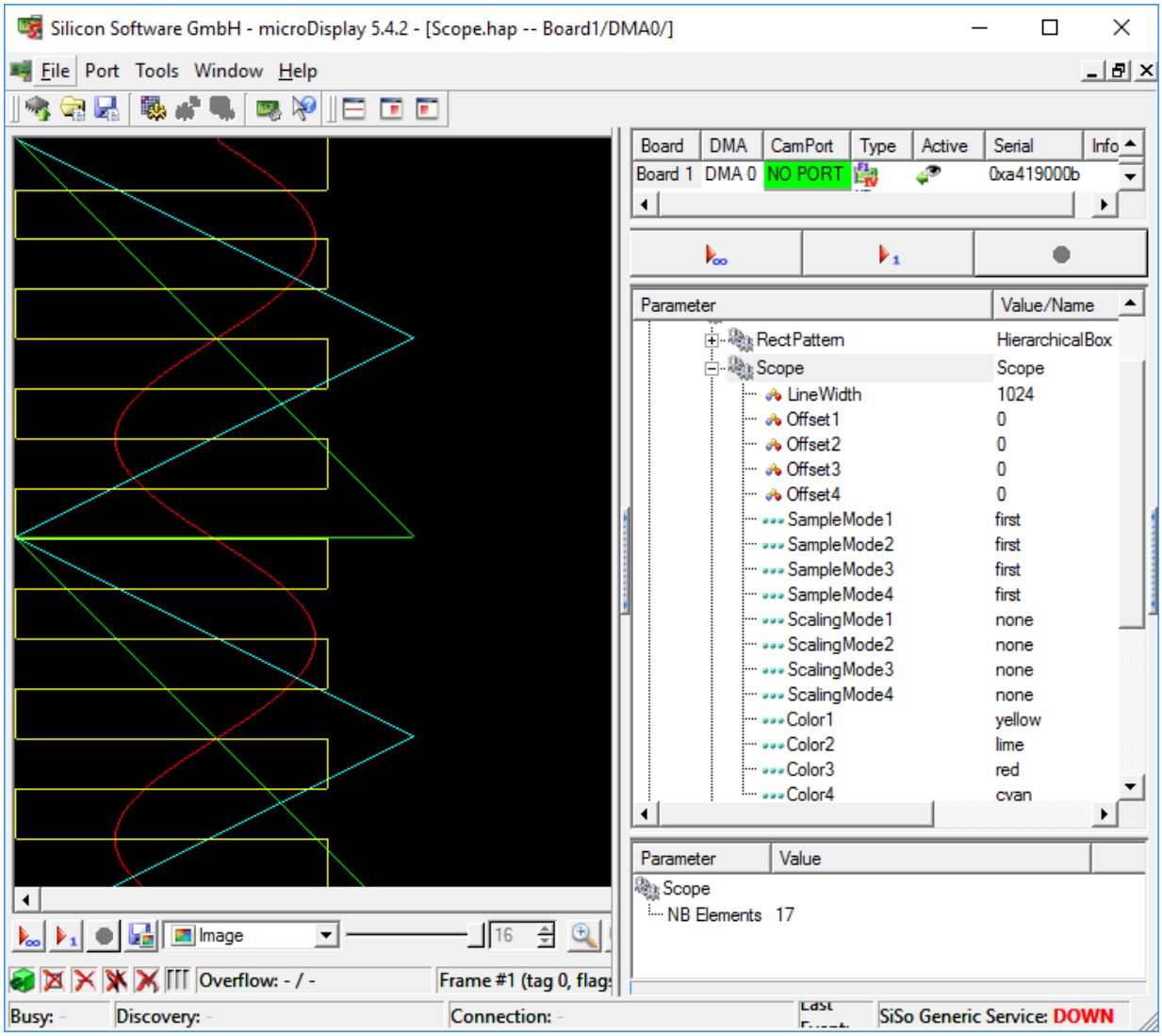



To create a waveform out of the individual pixels values, the values are connected via lines.

The height of the output waveform corresponds to the input image height.

This operator supports up to four channels by using the input link parallelism to separate the input channels. This means, up to four waveforms may be created at a time.

For each channel, i.e., for each waveform that is output, you can define a color value, an offset, and a scaling.




**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

The operator is not intended for design simulation within VisualApplets.

### 24.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, Image data input |
| **Output Link** | O, Image data output |

### 24.4.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | 24 |
| **Arithmetic** | {unsigned} | as I |
| **Parallelism** | [1, 4] | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D} | as I |
| **Color Format** | GRAY | VAF_Color |
| **Color Flavor** | None | FL_RGB |
| **Max. Img Width** | any | [8, 4096]❶ |
| **Max. Img Height** | any | as I |

❶ Please note that a configuration where the output image width is larger than the input image width can cause a slowdown of fetching input data.

### 24.4.3. Parameters

| **LineWidth** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [8, Maximum Output Image Width] |
| Defines the output line width. | |

| **Offset[1, 4]** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [-Maximum Output Image Width, Maximum Output Image Width - 1] |
| Horizontal position offset to be added to each output channel. | |

| **SampleMode[1, 4]** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | first |
| **Range** | {first, last, min, max} |
| Sets the sample mode used in each channel to acquire pixel value in each line.<br>**First**: value of the first pixel in line.<br>**Last**: value of the last pixel in line.<br>**Min**: value of the smallest pixel value found in line.<br>**Max**: value of the greatest pixel value found in line. | |

| **ScalingMode[1, 4]** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | none |
| **Range** | {none, div128, div64, div32, div16, div8, div4, div2, none, mult2, mult4, mult8, mult16, mult32, mult64, mult128} |

| ScalingMode[1, 4] |
| --- |
| Sets the waveform zoom mode for each channel. The captured pixel value can be multiplied or divided by multiples of two. |

| Color[1, 4] | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | blue |
| **Range** | {cyan, black, blue, green, cyan, red, magenta, brown, lightgray, darkgray, brightblue, brightgreen, brightcyan, brightred, brightmagenta, brightyellow, white} |
| Parameter for setting the output color for each channel. | |

### 24.4.4. Examples of Use

The use of operator Scope is shown in the following examples:

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

## 24.5. Operator ImageInjector

**Operator Library: Debugging**

The operator *ImageInjector* allows you to inject images into the image data output of the operator. The images can be injected directly from file (file format: *.tif). You can give the command for injecting via Framegrabber API or via the Framegrabber SDK tool microDisplay.



**Availability**

To use the *ImageInjector* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

The operator *ImageInjector* works like the *Simulation Sources* you use when simulating a design in VisualApplets - only that the operator *ImageInjector* is used for testing during runtime.

Two ways of injection are supported: insertion and replacement.

**Insert modes**: When one of the insert modes is active, the input link is blocked.

**Replacement modes**: When one of the replacement modes is active, the operator acts as an image sink during image injection. The input link is not blocked.

The injected image data is inserted pixel-by-pixel via writing to the *WritePixel* and *WriteFlag* registers. The write registers are only enabled, if the parameter *ReadyForInjection* is switched to "yes". (To understand when the parameter *ReadyForInjection* is set to "yes", see parameter *Mode* .)

At the end of each line of the injected image (except the last line), an *EndOfLine* flag must be written to the WriteFlag register. When the end of the last line is reached, an *EndOfFrame* flag must be written to the WriteFlag register.

The *CurrXPos* and *CurrYPos* parameters point to the currently active pixel position.



**The parameter *EnableInsertModes* changes the type of the operator from M-type to P-type**

The operator *ImageInjector* is an M-type operator. However, if you set *EnableInsertModes* to "no", the operator is handled like a P-Type operator by VisualApplets.



**Deactivating Injection Modes**

Switching back to default mode *FlowThrough* (i.e., deactivating all kinds of injection) is only allowed when the parameter *ReadyForFlowThrough* is set to "yes".



**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

The operator is not intended for design simulation within VisualApplets.

### 24.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, Image data input |
| **Output Link** | O, Image data output |

### 24.5.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 24.5.3. Parameters

| **Mode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | FlowThrough |
| **Range** | {FlowThrough, InsertAfterEof, InsertAfterEol, ReplaceAfterEof, ReplaceAfterEol} |

This mode parameter enables and disables image injection. During image injection, the operator either blocks the input link (InsertAfterEof, InsertAfterEol) or discards the image data that come in during injection (ReplaceAfterEof, ReplaceAfterEol).

**Inject**: In this mode, the operator immediately blocks the input link and sets the parameter *ReadyForInjection* to "yes". As soon as *ReadyForInjection* is set to "yes", you can inject an image from file.

**FlowThrough**: Image injection is disabled. Switching back to default mode FlowThrough (i.e., deactivating all kinds of injection) is only allowed when the parameter *ReadyForFlowThrough* is set to "yes".

**InsertAfterEof**: When set to this mode, the operator looks for an End-of-Frame (EoF) flag. As soon as it finds an EoF flag (i.e., when the end of the current image is reached), the operator blocks the input link and sets the parameter *ReadyForInjection* to "yes". As soon as *ReadyForInjection* is set to "yes", you can inject an image from file. This mode is not available with the LINE-1D image protocol. This mode is only available, if the parameter *EnableInsertModes* is set to "yes", which also implies that the operator is of type M.

**InsertAfterEol**: When set to this mode, the operator looks for an End-of-Line (EoL) flag. As soon as it finds an EoL flag (i.e., when the end of the current line is reached), the operator blocks the input link and sets the parameter *ReadyForInjection* to "yes". As soon as *ReadyForInjection* is set to "yes", you can inject an image from file. This mode is only available, if the parameter *EnableInsertModes* is set to "yes", which also implies that the operator is of type M.

When set to one of the two available relacement modes, the operator acts as an image sink during image injection.

**ReplaceAfterEof**: The operator waits for the current image to end. As soon as it finds an EoF flag, the operator sets the parameter *ReadyForInjection* to "yes". As soon as *ReadyForInjection* is set to "yes", you can inject an image from file. During injection, incoming image data are discarded. Not available with the LINE-1D image protocol.

**ReplaceAfterEol**: The operator waits for the current line to end. As soon as it finds an EoL flag, the operator sets the parameter *ReadyForInjection* to "yes". As soon as *ReadyForInjection* is set to "yes", you can inject an image from file. During injection, incoming image data are discarded.

| EnableInsertModes | |
|---|---|
| **Type** | static write parameter |
| **Default** | no |
| **Range** | {yes, no} |
| If set to "yes", the parameter *Mode* can be set to the insert modes **InsertAfterEof** or **InsertAfterEol**. In this case, the operator is of type M. If set to "no", these two insert modes are not allowed for the parameter *Mode* and the operator is of type P. | |

| ReadyForInjection | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | no |
| **Range** | {yes, no} |
| Indicates readiness to write a pixel or a flag when the parameter *Mode* is in insert or replace mode. It must be set to "yes" in order to enable the parameters *WritePixel* and *WriteFlag*. | |

| ReadyForFlowThrough | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | no |
| **Range** | {yes, no} |
| Indicates readiness to switch back to FlowThrough mode (of the parameter *Mode*).<br>The parameter is automatically set to "yes" when to parameter *WriteFlag*<br>• in IMAGE-2D mode an EndOfFrame is written, or<br>• in LINE-1D mode an EndOfLine is written. | |

| CurrXPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, MaxImageWidth-1] |
| Displays the current line position. | |

| CurrYPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, MaxImageHeight-1] |
| Displays the current image height position. | |

| WritePixel | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^BitWidth-1] |
| Pixel value. If 24 bit color format is used, the following bit mapping must be applied: Red = [0, 7], Green = [9,15], Blue = [16,23]. | |

| WriteFlag | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | EndOfLine |
| **Range** | {EndOfLine, EndOfFrame} |

| WriteFlag |
|---|
| You need to write an EndOfLine flag at the end of each injected line, except at the end of the last line of an image.<br><br>You need to write an EndOfFrame flag at the end of the last line of an injected image.<br><br>This parameter also influences the parameter *ReadyForFlowThrough*:<br><br>The parameter *ReadyForFlowThrough* is set to "yes", when<br><br>• in IMAGE-2D mode an EndOfFrame is written, or<br><br>• in LINE-1D mode an EndOfLine is written.<br><br>**Use only one flag at a time**<br><br>At the end of an image, only write an EndOfFrame flag to this parameter.<br><br>Do not write an EoL together with an EoF. EoF always includes EoL+EoF. |

| ImageFile | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | image.tif |
| **Range** | |
| Image file to be injected. The image dimensions must be equal or smaller than the ouput link properties. The pixel depth of the image file can be 8 bit or 16 bit per component. If the image pixel depth per component is not equal to the bit width per component defined by the link property, the image components are shifted so that they are most significant bit (MSB)-aligned to the pixel components defined by the link property. For the file selection in microDisplay, set the file path manually or right-click this parameter to open a file selection menu. | |

| InjectFromFile | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | no |
| **Range** | {no, yes} |
| When set to "yes", the image file is injected into the output link. Disabled in FlowThrough mode (i.e., when parameter *Mode* is set to FlowThrough). | |

### 24.5.4. Examples of Use

The use of operator ImageInjector is shown in the following examples:

- Section 11.4, 'Functional Example for Loading Test Images Using *ImageInjector* '

  Examples - Demonstration of how to use the operator

- Section 12.6.4, 'Manual Image Injection'

  Example - For debugging purposes images can be inserted manually.

## 24.6. Operator ImageTimingGenerator

**Operator Library: Debugging**

Operator *ImageTimingGenerator* allows you to emulate the image timing as it occurs with real cameras in productive systems.



**Availability**

To use the *ImageTimingGenerator* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

The operator creates a black image output. It offers many parameters that enable you to define the timing behavior of the image output.

**Use this operator to test whether the subsequent image processing pipeline can handle a specific timing of the data stream.**

You will see if a specific timing behavior causes bottlenecks, or if the pipeline of your design is able to process the images despite timing deviations.

The line generation settings and the frame generation settings support four different generation modes:

- FreeRun: The FreeRun mode achieves the fastest generation.
- JitterMode:The JitterMode works like the FreeRun mode but enables you to define jitter for timing generation.
- FixedFreqency: The FixedFreqency generation mode allows you to set a fix timing.
- ExternalTrigger: If you want to use an input signal for triggering the line and/or frame generation, you set the line and/or frame generation settings to ExternalTrigger mode.

In addition, you can test how the design reacts to variations in image height, image width, line gap, and frame gap. You can define these parameters via lookup tables and variable jitter settings.

You can also adjust the pixel duty cycle to simulate a slowdown of a virtual pixel clock.

ImageHeightJitter and FrameGapJitter are only active (can only be used) as long as FrameGenerationMode is set to JitterMode. Likewise, ImageWidthJitter and LineGapJitter are only active during LineGenerationMode = JitterMode.

Frame generation parameters as well as ImageHeight parameters are only available with the VALT_IMAGE2D image output protocol. Also all Line generation parameters as well as ImageWidth parameters are only available with the VALT_IMAGE2D or the VALT_LINE1D image output protocol.



**Behavior During Design Simulation (Before Build)**

During design simulation, only the parameters *ImageHeightLut*, *ImageWidthLut*, *MaxImageSequenceLength* and *ImageSequenceLength* are used. All other operator parameters and LUT entries of this operator have no influence on the simulated image. This means that during simulation, the operator *ImageTimingGenerator* behaves like the operator *CreateBlackImage*.

All other operator parameters and LUT entries of this operator have no influence on the simulated image (i.e., during simulation, operator *ImageTimingGenerator* behaves like operator *CreateBlackImage*).



**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

### 24.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | TriggerLineI (optional), optional external line trigger signal input. Tie to ground if not used. TriggerFrameI (optional), optional external frame trigger signal input. Tie to ground if not used. |
| **Output Link** | O, Image data output |

### 24.6.2. Supported Link Format

| Link Parameter | Input Link TriggerLineI (optional) | Input Link TriggerFrameI (optional) | Output Link O |
|---|---|---|---|
| **Bit Width** | 1 | 1 | 1 |
| **Arithmetic** | none | none | unsigned |
| **Parallelism** | none | none | any |
| **Kernel Columns** | none | none | 1 |
| **Kernel Rows** | none | none | 1 |
| **Img Protocol** | SIGNAL | SIGNAL | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} |
| **Color Format** | none | none | GRAY |
| **Color Flavor** | none | none | none |
| **Max. Img Width** | none | none | any |
| **Max. Img Height** | none | none | any |

### 24.6.3. Parameters

| **MaxImageSequenceLength** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | [1, (2^14)-1] |
| Defines the LUT size for ImageHeightLUT, ImageWidthLUT, FrameGapLUT, and LineGapLUT. | |

| **ImageSequenceLength** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1 |
| **Range** | [1, (2^MaxImageSequenceLength)-1 ] |
| Sets the number of entries used from the LUTs. If set to one, only the first entry of each LUT is used. If set to, e.g., four, while the MaxImageSequenceLength is set to, e.g., 7, only the first four entries in all LUTs will be used in round robin fashion. | |

| **ImageHeightLUT** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [1, MaxImageHeight] |
| This LUT stores image height values to be used for output image generation. During generation the LUT values are read in round robin fashion. | |

| ImageHeightJitter | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, MaxImageHeightBitwidth-2] |
| Number of least significant bits of an ImageHeightLUT value that will be allowed to jitter during image generation. Only used while parameter FrameGenerationMode is set to JitterMode. | |

| ImageWidthLUT | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [1, MaxImageWidth] |
| This LUT stores image width values to be used for output image generation. During generation the LUT values are read in round robin fashion. | |

| ImageWidthJitter | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, MaxImageWidthBitwidth-2] |
| Number of least significant bits of an ImageWidthLUT value that will be allowed to jitter during image generation. Only used while parameter LineGeneration Mode is set to JitterMode. | |

| PixelDutyCycle | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 100 |
| **Range** | [1, 100] |
| Pixel duty cycle for the output stream. 100 % means that every clock cycle corresponds to an output pixel. If 20% pixel duty cycle is set then every fifth clock cycle an output pixel is generated. | |

| FrameGenerationMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | FreeRun |
| **Range** | {FreeRun, JitterMode, FixedFrequency, ExternalTrigger} |

Selects the frame generation mode.

**FreeRun**: Fastest image generation mode based on ImageHeight and FrameGap. Parameter FrameRate is ignored.

**JitterMode**: Same as FreeRun but using the FrameGapJitter parameter.

**FixedFrequency**: Generation of a fixed frequency image rate set through the FrameRate parameter.

**ExternalTrigger**: Optional external trigger input TriggerFrameI is used to trigger the start of a new image frame generation. Only available if port FrameTriggerI was enabled.

When a new image start is triggered before a current image generation has ended, this request is ignored an the SkippedFrame counter is incremented. This applies to all frame generation modes.

| FrameRate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 10 |
| **Range** | [0.5, 1048576.0] |

| FrameRate |
|---|
| Target image frequency in Hz. Only used when parameter FrameGenerationMode is set to FixedFrequency. Maximum FrameRate is limited by the LineRate and the ImageHeight. |

| FrameGapLUT | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [1, 67108863] |
| Lookup table entries define the gap between two image frames. The frame gap value defines the gap as number of clock cycles. This parameter is ignored in FixedFreqency generation mode . | |

| FrameGapJitter | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, FrameGapBitwidth-2] |
| Number of least significant bits of an FrameGapLUT value that will be allowed to jitter during image generation. Only used during JitterMode frame generation. | |

| LineGenerationMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | FreeRun |
| **Range** | {FreeRun, JitterMode, FixedFrequency, ExternalTrigger} |

Selects the line generation mode.

**FreeRun**: Fastest image generation mode based on ImageWidth and LineGap. Parameter LineRate is ignored.

**JitterMode**: Same as FreeRun except the use of LineGapJitter parameter.

**FixedFrequency**: Generation of a fixed frequency line rate set through the LineRate parameter.

**ExternalTrigger**: Optional external trigger input TriggerLineI is used to trigger the start of a new line generation. Only available if port LineTriggerI was enabled.

If current line generation has not ended and a new line start is triggered then this request is ignored an the SkippedLine counter is incremented. This applies to all line generation modes.

| LineRate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 10 |
| **Range** | [0.5, 1048576.0] |
| Target line frequency in Hz. Only used in FixedFrequency line generation mode. Maximum LineRate is limit by the LineGap and the ImageWidth. | |

| LineGapLUT | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [1, 67108863] |
| Lookup table entries define the gap between two lines. The line gap value defines the gap as number of pixels between two lines. This parameters is ignored in FixedFreqency line generation mode. | |

| LineGapJitter | |
|---|---|
| **Type** | dynamic write parameter |

| LineGapJitter | |
|---|---|
| **Default** | 0 |
| **Range** | [0, LineGapBitwidth-2] |
| TNumber of least significant bits of an LineGapLUT value that will be allowed to jitter during image generation. Only used during JitterMode line generation. | |

| SkippedLine | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 65535] |
| This is an error counter used to indicate conflicting settings in the line generation mode.<br><br>If current line generation has not ended and a new line start is triggered then this request is ignored an the SkippedLine counter is incremented. This applies to all line generation modes. | |

| SkippedFrame | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 65535] |
| This is an error counter used to indicate conflicting settings in the frame generation mode.<br><br>If current image generation has not ended and a new image start is triggered then this request is ignored an the SkippedFrame counter is incremented. This applies to all frame generation modes. | |

| LinesToSimulate | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | [1, (2^32) - 1] |
| If the output link image protocol is set to VALT_LINE1D, it is required to specify the number of lines for simulation. This parameter is used to specify the simulation image height. See 'Simulation' for more information on 1D simulations. This parameter can only be changed if the output image protocol is set to VALT_LINE1D. | |

| PixelsToSimulate | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | [1, (2^32) - 1], step size = output parallelism |
| If the output link image protocol is set to VALT_PIXEL0D, it is required to specify the size of the pixel stream for simulation. This parameter is used to specify the simulation pixel stream width. See 'Simulation' for more information on 0D simulations. This parameter can only be changed if the output image protocol is set to VALT_PIXEL0D. | |

### 24.6.4. Examples of Use

The use of operator ImageTimingGenerator is shown in the following examples:

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

- Section 12.6.7, 'Image Flow Control'

  Example - For debugging purposes of the designs internal data flow control in hardware and a possible compensation.

# 24.7. Operator ImageFlowControl

**Operator Library: Debugging**

With operator *ImageFlowControl*, you can control the data flow that is received from infinite sources.



**Availability**

To use the *ImageFlowControl* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

Operator *ImageFlowControl* is especially helpful if your design

- is fed by unstoppable (infinite) sources and does not make use of RAM-based image buffers.
- contains loops and you want to detect and prevent data loss in loops.

Using this operator, you make sure that no overflows of FIFOs occur in the following pipeline.

The operator works similar to operator *ImageFifo* - it has a built-in FIFO.

In addition, the operator provides a specific overflow control mechanism. In case of overflow

- Images are cut (and only the first part of the image is forwarded).
- Images are discarded as long as the overflow situation is there. (When the overflow situation is over, the transfer starts with the first pixel of the next frame.)

This way, the operator ensures the design can be run.

Specific count parameters inform you how many frames were lost and how many frames were forwarded incomplete due to cutting them in an overflow situation. This allows you to track the data loss in your design (or, for example, in a loop).

Additionally, lost frames can be compensated with dummy frames of selectable width and height. This feature allows to keep the rate of incoming and outgoing frames constant in case of a temporal bandwidth restriction at the output link. However, this will only work as long as the generated dummy frames are small enough in size compared to the lost frames.



**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

The operator is not intended for design simulation within VisualApplets.

## 24.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | TerminateI (optional), external terminate signal input. If set to one, the current input image is terminated, i.e., the operator acts as a sink. Compensation of lost frames can be enabled through the CompensateLostFrames parameter.<br>I, image data input |
| **Output Links** | O, image data output |

| Property | Value |
|---|---|
| | StatusO (optional), statusO port is synchronous to image data output port O. This port is two bit wide. Bit 0: Signals truncation of the current frame if set. Bit 1: Signals a dummy frame if set. Only the last data word inside a frame must be evaluated. |

### 24.7.2. Supported Link Format

| Link Parameter | Input Link TerminateI (optional) | Input Link I |
|---|---|---|
| **Bit Width** | 1 | [1, 64] |
| **Arithmetic** | none | {unsigned, signed} |
| **Parallelism** | none | any |
| **Kernel Columns** | none | any |
| **Kernel Rows** | none | any |
| **Img Protocol** | SIGNAL | VALT_IMAGE2D |
| **Color Format** | none | any |
| **Color Flavor** | none | any |
| **Max. Img Width** | none | any |
| **Max. Img Height** | none | any |

| Link Parameter | Output Link O | Output Link StatusO (optional) |
|---|---|---|
| **Bit Width** | as I | 2 |
| **Arithmetic** | as I | as I |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as I |
| **Kernel Rows** | as I | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | GRAY |
| **Color Flavor** | as I | none |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

### 24.7.3. Parameters

| EntitiesToStore | |
|---|---|
| **Type** | static write parameter |
| **Default** | 8 |
| **Range** | [1, (2^32)-1] |
| Number of fifo entities to store. | |

| EntitiesType | |
|---|---|
| **Type** | static write parameter |
| **Default** | LINE |
| **Range** | {FRAME, LINE, PIXEL} |

| EntitiesType |
| --- |
| Defines the storage entity. |

| ImplementationType | |
| --- | --- |
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | {AUTO, BRAM, LUTRAM} |
| Defines the fifo implementation type. | |

| FifoFillLevel | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | 0% |
| **Range** | [0%, 100%] |
| Actual fifo fill level in percent. | |

| MaxFifoFillLevel | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | 0% |
| **Range** | [0%, 100%] |
| Stores the maximum fifo fill level ever reached. | |

| CounterWidth | |
| --- | --- |
| **Type** | static write parameter |
| **Default** | 16 |
| **Range** | [8, 32] |
| Sets the bit width of all counters inside this operator. | |

| LostFrameCount | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 16 |
| **Range** | [0, (2^CounterWidth)-1] |
| Counts whole frames which were lost due blocking of the output link or the optional TerminateI input.ra> | |

| IncompleteFrameCount | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 16 |
| **Range** | [0, (2^CounterWidth)-1] |
| Counts incomplete frames which where prematurely terminated. | |

| FullFrameCount | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 16 |
| **Range** | [0, (2^CounterWidth)-1] |
| Counts full frames which were passed through to the output link. | |

| ClearStatus | |
| --- | --- |
| **Type** | dynamic write parameter |

| ClearStatus | |
|---|---|
| **Default** | No |
| **Range** | {Yes, No} |
| If set to "yes" all counters are cleared. | |

| CompensateLostFrame | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {Yes, No} |
| If set to Yes all lost frame are compensated through the insertion of dummy frames. | |

| DummyFrameWidth | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [2, MaxImageWidth] |
| Width of the dummy frame to be generated. | |

| DummyFrameHeight | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1 |
| **Range** | [1, MaxImageHeight] |
| Height of the dummy frame to be generated. | |

| DummyFrameCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0,(2^CounterWidth)-1] |
| Counts all generated dummy frames. | |

| EofDifferenceFrameCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [-(2^CounterWidth), (2^CounterWidth)-1] |
| Difference between the number of outgoing and incoming frames: OutgoingEofCounter-IncomingEofCounter. | |

### 24.7.4. Examples of Use

The use of operator ImageFlowControl is shown in the following examples:

- Section 12.6.7, 'Image Flow Control'

  Example - For debugging purposes of the designs internal data flow control in hardware and a possible compensation.

# 24.8. Operator StreamControl

**Operator Library: Debugging**

This operator you can use to manipulate the blocking behavior on the image link. You can suppress the blocking status as well as create blocking pulses towards the input link.



**Availability**

To use the *StreamControl* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

**Blocking:** You can use the operator to test how the design reacts to blocking situations.

**Suppressing:** When you suppress a blocking situation, you can test the preceding part of the design even though the subsequent parts cause a blocking situation. Those subsequent parts of the pipeline will typically not work properly due to data loss. Data loss is monitored by parameter *LostDataWords*.

Example: If you place the operator directly in front of the *DmaToPC* operator and suppress blocking, you can find out about the maximum speed of your design (since you suppress limitations merely due to the data width of a potential DMA bus).

You can use external input signals to control the blocking behavior.



**Data Loss while Blocking is Supressed**

While an inhibit (blocking) is in suppressed state, all incoming data is lost since the output link is not ready to accept new data.



**Use Carefully**

This operator must be used with caution as it can disturb the inherent handshaking protocol between operators.



**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

The operator is not intended for design simulation within VisualApplets.

## 24.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, Image data input<br>ExtI, external signal input (optional) |
| **Output Link** | O, Image data output |

## 24.8.2. Supported Link Format

| Link Parameter | Input Link I | Input Link ExtI | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] | 1 | as I |
| **Arithmetic** | {unsigned, signed} | none | as I |

| Link Parameter | Input Link I | Input Link ExtI | Output Link O |
|---|---|---|---|
| **Parallelism** | any | none | as I |
| **Kernel Columns** | any | none | as I |
| **Kernel Rows** | any | none | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | SIGNAL | as I |
| **Color Format** | any | none | as I |
| **Color Flavor** | any | none | as I |
| **Max. Img Width** | any | none | as I |
| **Max. Img Height** | any | none | as I |

### 24.8.3. Parameters

| **ClearMode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | ClearWithProcessStart |
| **Range** | {NoClearing, ClearWithProcessStart, ClearWithProcessReset, SendClearPulse} |

This parameter defines the reset behavior for all **read** parameters.

**NoClearing**: Values of all *read* parameters are held. ProcessEnable and ProcessReset have no influcence on these values.

**ClearWithProcessStart**: Values of all *read* parameters are only cleared with rising edge of ProcessEnable.

**ClearWithProcessReset**: Values of all *read* parameters are only cleared with rising edge of ProcessReset.

**SendClearPulse**: On-demand clearing of all *read* parameter values.

| **InhibitMode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | FlowThrough |
| **Range** | {FlowThrough, OverridePulseHi, OverridePulseLow, OverrideExtHi, OverrideExtLow, PulsHi, PulsLow, StaticHi, StaticLow, Ext} |

**FlowThrough**: Inhibit (blocking) passes from O to I.

**OverridePulseHi**: Creates a one clock cycle wide Inhibit (blocking) signal towards I when selected. Otherwise, same behavior as in FlowThrough mode.

**OverridePulseLow**: Suppresses the Inhibit (blocking) signal for one clock cycle. Otherwise, same behavior as in FlowThrough mode.

**OverrideExtHi**: Only high level input from external signal input ExtI is passed as an Inhibit (blocking) to I. Otherwise, same behavior as in FlowThrough mode.

**OverrideExtLow**: Only low level input from external signal input ExtI is used to suppress Inhibit (blocking) towards I. Otherwise, same behavior as in FlowThrough mode.

**PulsHi**: Creates exactly one Inhibt (blocking) cycle while Inhibit (blocking) is set permanently to zero.

**PulsLow**: Supresses exactly one Inhibt (blocking) cycle while Inhibit (blocking) is set permanently.

**StaticHi**: Forces Inhibt (blocking) to be active permanently.

**StaticLow**: Deactivates Inhibt (blocking) permanently.

| InhibitMode |
| --- |
| **Ext**: ExtI input signal sets the Inhibit (blocking) state directly. |

| LostDataWords | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | |
| Counts image data that are lost due to suppressed Inhibit (blocking). | |

| GeneratedInhibits | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | |
| Counts generated Inhibits (blockings) while no Inhibit (blocking) was present at ouput O. | |

# 24.9. Operator ImageMonitor

**Operator Library: Debugging**

Operator *ImageMonitor* fetches images for you from any spot of the inner pipeline you define. You can see what happened to the image until it reached this spot, and use this information for debugging.



**Availability**

To use the *ImageMonitor* operator, you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

Operator *ImageMonitor* works like a *Simulation Probe* you use when simulating a design in VisualApplets - only that operator *ImageMonitor* is used for testing during runtime.

The operator provides a very simple image readout register interface. When parameter Mode is set to image monitoring, whole images can be read by only reading the PixelData register.

This is very helpful as you can monitor image data without using one of the (limited) DMA channels.

The number of *ImageMonitor* operator instances in a design is theoretically unlimited. However, you should use the operator carefully to save resources as additional FPGA logic is introduced for pausing the data flow during read-out. This also means that the operator cannot be used for monitoring infinite sources.

Supported pixel width is limited to 61 bits since the last three bits of PixelData carry the Valid, EndOfLine, and EndOfFrame flags.

The CurrXPos and CurrYPos parameters point to the currently active pixel position. This information can be used to filter the output pixels, e.g., to analyze only each second or third pixel.



**Runtime Testing**

This operator is designed for testing and analyzing your design during runtime: You need to build (synthesize) the design, load it onto the target hardware, and start actual image processing, before you can use the operator for debugging.

The operator is not intended for design simulation within VisualApplets.

## 24.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |

## 24.9.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | [1, 61] |
| **Arithmetic** | {unsigned, signed} |
| **Parallelism** | any |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} |
| **Color Format** | any |

| Link Parameter | Input Link I |
|---|---|
| **Color Flavor** | any |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 24.9.3. Parameters

| Mode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | UnlimitedSink |
| **Range** | {UnlimitedSink, Monitor, MonitorFromLineStart, MonitorFromFrameStart} |
| This parameter sets the working mode.<br>**UnlimitedSink** = No monitoring as all input is discarded.<br>**Monitor** = Switches to monitoring mode immediately.<br>**MonitorFromLineStart** = Switches to monitoring mode after the end of current line.<br>**MonitorFromFrameStart** = Switches to monitoring mode after the end of current frame. | |

| ReadyForMonitoring | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | no |
| **Range** | {yes, no} |
| Indicates readiness to read PixelData parameter. PixelData must be read only if set to yes. | |

| PixelData | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | [1, 64] |
| Includes the current pixel value and three pixel flags. When valid flag is not set, the pixel value is discarded.<br>PixelValue = PixelData[1, LinkBitWidth]<br>ValidFlag = PixelData[LinkBitWidth+1]<br>EndOfLineFlag =<br>PixelData[LinkBitWidth+2]<br>EndOfFrameFlag =<br>PixelData[LinkBitWidth+3] | |

| CurrXPos | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, MaxImageWidth-1] |
| Displays current line position. | |

| CurrYPos | |
|---|---|
| **Type** | dynamic read parameter |

| CurrYPos | |
|---|---|
| **Default** | 0 |
| **Range** | [0, MaxImageHeight-1] |
| Displays current image height position. | |

### 24.9.4. Examples of Use

The use of operator ImageMonitor is shown in the following examples:

- Section 12.6.5, 'Image Monitoring'

  Example - For debugging purposes image transfer states on links can be investigated.

# 25. Library Filter




The *Filter* library includes filter operators for any kind of image filters and kernel operations.

The following list summarizes all Operators of Library Filter

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | DILATE | Performs a binary dilation of the image, which can be used to close gaps in images. | Version 1.1 |
|  | ERODE | Performs a binary erosion of the image, which can be used to separate touching objects. | Version 1.1 |
|   | FIRkernelNxM | Creates rectangular kernel window around each pixel, which includes the neighbor pixels. | Version 1.1 |
|   | FIRoperatorNxM | Calculates the sum of the multiplication of the input kernel elements with parameterizable coefficients. | Version 1.1 |
|  | HitOrMiss | Implements the morphological image processing operator hit-or-miss as matching element with defined structure. | Version 1.2 |
|  | LineNeighboursNx1 | Creates a kernel window of N rows and 1 column. | Version 1.2 |
|  | MAX | Identifies and outputs the maximum value of an input kernel-stream. | Version 1.1 |
|  | MEDIAN | Identifies and outputs the median of an input kernel-stream. | Version 1.1 |
|  | MIN | Identifies and outputs the minimum value of an input kernel-stream. | Version 1.1 |
|  | NumberOfHits | Counts the number of matches of the input kernel with a defined structure. | Version 1.2 |
|  | PixelNeighbours1xM | Creates a kernel window of 1 row and M columns. | Version 1.2 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | SORT | Sorts all elements of an input kernel-stream by their gray values. | Version 1.2 |

Table 25.1. Operators of Library Filter

# 25.1. Operator DILATE

**Operator Library: Filter**

The operator DILATE performs a binary dilation of the image, which can be used to close gaps in images. The input of the operator is a kernel image data stream. At the output, the dilation result of the central pixel is provided. A structuring element for the dilate operation is given by the parameterizable binary matrix *StructElement*. The matrix has the size of the input kernel. Dilation is a fundamental operation in morphological image processing. The operator output is set to '1', if any matrix element of value '1' matches with its corresponding kernel element.

## 25.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, result output |

## 25.1.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 25.1.3. Parameters

| **StructElement** | |
|---|---|
| **Type** | static parameter |
| **Default** | 1 |
| **Range** | {0, 1} |

Structuring element is a shape, used to probe or interact with a given image, with the purpose of drawing conclusions on how this shape fits or misses the shapes in the image.

- A '0' in the struct element has to match with a '0' in the respective kernel input.
- A '1' in the struct element has to match with a '1' in the respective kernel input.

If the input kernel size of the operator is changed, the coefficients will also change. Check the coefficients after changing the input kernel size.

## 25.1.4. Examples of Use

The use of operator DILATE is shown in the following examples:

- Section 12.8.2.1, 'Close'

  Examples - Shows the implementation of a morphological close applied to binary images.

- Section 12.8.2.3, 'Open'

  Examples - Shows the implementation of a morphological open applied to binary images.

# 25.2. Operator ERODE

**Operator Library: Filter**

The operator ERODE performs a binary erosion of the image, which can be used to separate touching objects. The input of the operator is a kernel image data stream. At the output the erosion result of the central pixel is provided. A structuring element for the erode operation is given by a parameterizable binary matrix *StructElement*. The matrix has the size of the input kernel. Erosion is a fundamental operation in morphological image processing. In erosion, the output is set to '1', if all matrix elements of value '1' match with their corresponding kernel elements. In other words, the kernel matrix has to fully fit into the image.

## 25.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, result output |

## 25.2.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 25.2.3. Parameters

| **StructElement** | |
|---|---|
| **Type** | static parameter |
| **Default** | 1 |
| **Range** | {0, 1} |

Structuring element is a shape, used to probe or interact with a given image, with the purpose of drawing conclusions on how this shape fits or misses the shapes in the image.

- A '0' in the struct element has to match with a '0' in the respective kernel input.
- A '1' in the struct element has to match with a '1' in the respective kernel input.

If the input kernel size of the operator is changed, the coefficients will also change. Check the coefficients after changing the input kernel size.

### 25.2.4. Examples of Use

The use of operator ERODE is shown in the following examples:

- Section 12.8.1.1, 'Morphological Edge'

  Examples - A binary eroded image is compared with the original. An edge is detected if both differ.

- Section 12.8.2.1, 'Close'

  Examples - Shows the implementation of a morphological close applied to binary images.

- Section 12.8.2.3, 'Open'

  Examples - Shows the implementation of a morphological open applied to binary images.

## 25.3. Operator FIRkernelNxM

**Operator Library: Filter**

The operator *FIRkernelNxM* creates rectangular kernel window around each pixel, which includes the neighbor pixels. The size of the kernel can be defined by editing the output link. *N* represents the number of kernel rows and *M* the number of kernel columns.



**No FIR filter**

In contrast to its name, the operator has nothing to do with a FIR filter. It can be used to produce the filter input data.

Other operators to generate kernels are the operators *LineNeighboursNx1* and *PixelNeighbours1xM*.

The central element of a kernel always represents the original input pixel of the respective position. The output image O with pixel indices x and y and kernel indices m and n is derived from the input by:

$$O(x,y,m,n) = I(x+m^*, y+n^*)$$

with $m^* = m - \left\lceil \frac{M}{2} \right\rceil + 1$ and $n^* = n - \left\lceil \frac{N}{2} \right\rceil + 1$

At image edges, no neighbored pixels are defined. The operator allows the definition of the required edge handling algorithm using the parameter *EdgeHandling*.

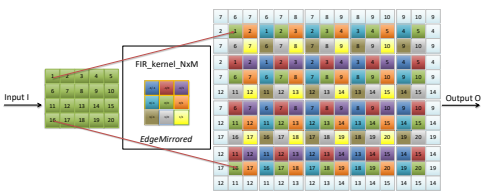
For kernels of even size, there is no central pixel. In this case, the central pixel is displaced to the upper left corner, e.g. for a kernel of size 4x4 defines the central pixel at (1,1) as can be seen in the following table.

| | | | |
|---|---|---|---|
| 0,0 | 0,1 | 0,2 | 0,3 |
| 1,0 | **1,1** | 1,2 | 1,3 |
| 2,0 | 2,1 | 2,2 | 2,3 |
| 3,0 | 3,1 | 3,2 | 3,3 |

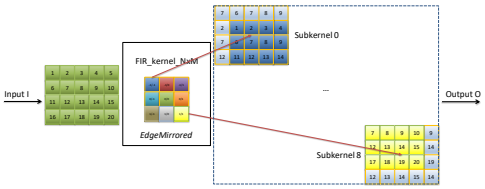
Kernels are described by the number of rows and columns. In VisualApplets, *N* represents the number of rows, and *M* represents the number of columns. Often, kernel elements are addressed by their index instead of their coordinate. In this case, the order is row by row as shown in the following table.

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | **5** | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

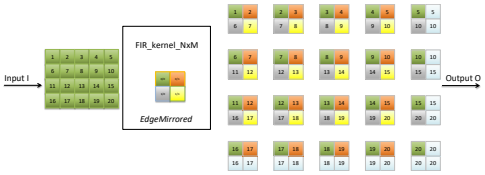
The following example shows the resulting composition for three kernel rows and three kernel columns. The 3x3 matrix in the depicted operator provides the relative addressing that defines the content of the respective kernels. Each 3x3 cluster on the output shows the different kernel values per pixel.



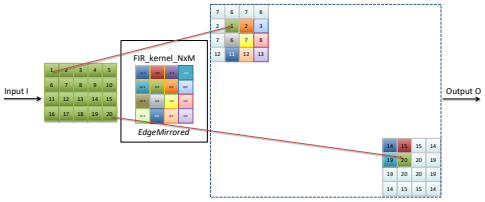

Instead of kernel configurations per pixel, this example shows the resulting subkernels: the first kernel at index 0 and the last kernel at index 8.




The following example shows the resulting composition for two kernel rows and two kernel columns. As the relative addresses only reference neighbors on the right hand side or below, no edge handling needs to be performed for content to the left or at top.




Comparing the last two examples with odd and even kernel sizes shows a noticeable behavior of the edgeHandling feature in *edgeMirroring* mode. While *edgeMirroring* for even and odd kernel sizes is treated identically on top and left borders, it produces slightly different results on the bottom and the right hand side. Due to the large output image size only the resulting configurations for the first, upper right, and last, lower right, input pixel with their resulting values per kernel are presented:




*EdgeMirroring* for odd kernel sizes fills regions with missing data from equidistant regions with data. The border is preserved and not replicated. The same principle is valid for filling regions to the top or to the left for even kernel sizes. In contrast to this, filling regions to the bottom or to the right replicates the border during mirroring. Again, this only occurs for even kernel sizes.



***EdgeMirroring* for even kernel sizes**

For even kernel sizes, filling regions to the bottom or to the right replicates the border during mirroring.

Technically, the operator waits and buffers the required pixels of the kernel until they can be output in parallel. Therefore, the operator will cause a delay as pixels can only be output after all required respective neighbor pixels were processed at the input. The line delay is:

$$LineDelay = \lfloor KernelRows/2 \rfloor$$

**Operator Restrictions**

- ImageWidth < 2*Parallelism is not allowed.
- Empty images, i.e. images with no pixels, are not allowed.
- Empty lines or varying line lengths are not allowed.

## 25.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |

| Property | Value |
|---|---|
| **Input Link** | I, data input |
| **Output Link** | O, kernel output |

## 25.3.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | any❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | {1, 2, 4, 8, 16, 32, 64} | as I |
| **Kernel Columns** | 1 | any❷ |
| **Kernel Rows** | 1 | any❸ |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D}❹ | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

❹ If VALT_LINE1D image protocol is used, the operator is of type P instead of type M.
❷ OutputKernelColumns <= ImageWidth
❸ OutputKernelRows <= ImageHeight

## 25.3.3. Parameters

| EdgeHandling | |
|---|---|
| **Type** | static parameter |
| **Default** | EdgeMirrored |
| **Range** | {EdgeMirrored, EdgeConstant} |
| At image edges, no neighbors exist to generate a kernel. In this case, the operator can be used in two edge handling modes:<br><br>In mode *EdgeMirrored*, mirrored pixels are used at the image edges.<br><br>In mode *EdgeConstant*, the missing kernel elements at the edges are filled by a constant value. This value can be defined using the parameter *Constant*. | |

| Constant | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | range of input link I |
| If the parameter *EdgeHandling* is set to *EdgeConstant*, this parameter defines the default value for a pixel outside the image borders. | |

| Constant |
| --- |
| For color formats, the value is an unsigned integer combining the binary representations of all color components. |

### 25.3.4. Examples of Use

The use of operator FIRkernelNxM is shown in the following examples:

- Section 3.6.6, 'Timing Synchronization'

  Synchronization - Avoiding deadlocks.

- Section 12.1.6, 'Laser Pointer Detection'

  Examples - A convolution with high intensity spot coefficients is made. For results above threshold, the respective pixels are dyed in red.

- Section 12.2.1, 'Adaptive Threshold'

  A binarization example for local adaptive thresholding. A kernel size of 8 by 8 pixel is used.

- Section 12.4.1.1, 'Nearest Neighbor De-Mosaicing'

  Examples - Nearest Neighbor Bayer De-Mosaicing

- Section 12.4.1.2, 'Bayer 3x3 De-Mosaicing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter.

- Section 12.4.1.3, 'Bayer 5x5 De-Mosaicing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter.

- Section 12.4.1.4, 'Bayer 3x3 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 3x3 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.5, 'Bayer 5x5 De-Mosacing with White Balancing'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern using a 5x5 filter. Moreover, a white balancing for color correction is added.

- Section 12.4.1.6, 'Edge Sensitive Bayer De-Mosaicing Algorithm'

  Examples - Edge Sensitive Laplace Bayer De-Mosaicing filter

- Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche'

  Examples - Laroche Bayer De-Mosaicing filter

- Section 12.4.1.8, 'Modified Laroche Bayer De-Mosaicing Algorithm '

  Examples - Ressource Optimized Laroche Bayer De-Mosaicing filter

- Section 12.4.1.9, 'Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern Red/BlueFollowedByGreen GreenFollowedByBlue/Red '

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a bilinear line scan camera with color pattern Red/BlueFollowedByGreen_GreenFollowedByBlue/Red

- Section 12.4.1.10, 'Bayer DeMosaicing For Bilinear LineScan Cameras with Color Pattern RedFollowedByBlue GreenFollowedByGreen '

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a bilinear line scan camera with color pattern Red/BlueFollowedByBlue/Red_GreenFollowedByGreen

- Section 12.4.1.11, 'Bayer DeMosaicing a Line Scan Camera with Bi-color Bayer Pattern biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR'

  Examples - The example shows the de-mosaicing of a Bayer RAW pattern of a CXP-12 line scan camera with bi-color Bayer pattern: biColorRGBG, biColorGRGB, biColorBGRG and biColorGBGR, for example the racer 2 L camera. In addition, the example contains a line scan trigger module and a white balancing module.

- Section 12.5.1, 'Co-Processor Median Filter'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a median filter is calculated.

- Section 12.5.2, 'Co-Processor Large Filter Calculation'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a large filter kernel is calculated.

- Section 12.8.1.1, 'Morphological Edge'

  Examples - A binary eroded image is compared with the original. An edge is detected if both differ.

- Section 12.8.1.2, 'Kirsch Filter'

  Examples - The Kirsch filter is a good edge detection filter for non directional edges.

- Section 12.8.1.4, 'Sobel Gradient X'

  Examples - A Sobel filter in x-direction only.

- Section 12.8.1.5, 'Sobel Multi Gradient'

  Examples - A Sobel filter in all 4 directions.

- Section 12.8.2.1, 'Close'

  Examples - Shows the implementation of a morphological close applied to binary images.

- Section 12.8.2.3, 'Open'

  Examples - Shows the implementation of a morphological open applied to binary images.

- Section 12.8.3.1, 'Averaging 3x3'

  Examples - A simple 3x3 box filter.

- Section 12.8.3.2, 'Gaussian Filter 5x5'

  Examples - A Gauss filter using a 5x5 kernel.

- Section 12.8.3.3, 'Median Filter 5x5'

  Examples - Applet applies a 5x5 median filter on the image.

- Section 12.8.4.1, 'Filter Basics'

  Examples - Explains the implementation of filters.

- Section 12.8.4.2, 'Parallel Filters'

  Examples - An example of the use of two filters in parallel.

- Section 12.8.4.3, 'Filter Sub Kernels'

  Examples - Shows how to extract a sub kernel from a filter to obtain the original image data. This example performs a simple local adaptive binarization.

- Section 12.8.4.4, 'Filter for Line Scan Cameras'

Examples - Explains how to implement a filter for line scan cameras.

- Section 12.8.5.1, 'High Boost Sharpening Filter'

  Examples - A high boost Laplace filter for sharpening

- Section 12.8.5.2, 'Laplace Filter 3x3'

  Examples - A 3x3 Laplace filter.

- Section 12.12.2, 'Depth From Focus Using Loops'

  Examples - Depth From Focus using Loops

- Section 12.13.4, 'Normalized Cross Correlation'

  Examples-

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

# 25.4. Operator FIRoperatorNxM

**Operator Library: Filter**

The operator FIRoperatorNxM calculates the sum of the multiplication of the input kernel elements with parameterizable coefficients. The coefficients can be defined using parameter *Coefficients*.

Lets have a look at an example. Assume the following Sobel filter coefficients H to detect vertical edges

$$H(i,j) = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

This filter is applied to the following image I

$$I(x,y) = \begin{bmatrix} 20 & 25 & 32 & 89 & 103 & 120 \\ 25 & 27 & 32 & 87 & 108 & 126 \\ 25 & 29 & 31 & 85 & 101 & 115 \\ 18 & 23 & 36 & 85 & 108 & 112 \\ 21 & 25 & 35 & 89 & 102 & 120 \\ 25 & 29 & 38 & 77 & 95 & 99 \end{bmatrix}$$

For the pixel at position I(2,2) = 31 we get the following result at output O

$$O(2,2) = \sum_{i=0}^{i=2} \sum_{j=0}^{j=2} I(2-1+i, 2-1+j) * H(i,j) = 27 * -1 + 32 * 0 + 87 * 1 + 29 * -2 + 31 * 0 + 85 * 2 + 23 * -1 + 36 * 0 + 85 * 1 = 235$$

To generate the required input kernel use operators such as *FIRkernelNxM*, *LineNeighboursNx1* or *PixelNeighbours1xM*.

**Operator Restrictions**

- ImageWidth < 2*Parallelism are not allowed
- Empty images i.e. images with no pixels are not allowed.
- Empty lines or varying line lengths are not allowed.

## 25.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, data output |

## 25.4.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 63] signed | auto❶ |
| **Arithmetic** | {unsigned, signed} | auto❷ |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The required output bit width is automatically determined from the input bit width, arithmetic and coefficients.

The output bit width must not exceed 64 bit.

❷ The output arithmetic is automatically determined from the input arithmetic and the coefficients. The output is signed if either the input is signed or at least one coefficient is signed.

## 25.4.3. Parameters

| Coefficients | |
|---|---|
| **Type** | static parameter |
| **Default** | identity |
| **Range** | $\left[-2^{15}, 2^{15}-1\right]$ |
| This parameter defines the coefficients of the filter kernel. Signed and unsigned integer values are allowed.<br><br>If the input kernel size of the operator is changed, the coefficients will also change. Check the coefficients after changing the input kernel size.<br><br>The coefficient values may not cause the output bit width to exceed 64 bit. | |

## 25.4.4. Examples of Use

The use of operator FIRoperatorNxM is shown in the following examples:

- Section 12.1.6, 'Laser Pointer Detection'

  Examples - A convolution with high intensity spot coefficients is made. For results above threshold, the respective pixels are dyed in red.

- Section 12.2.1, 'Adaptive Threshold'

  A binarization example for local adaptive thresholding. A kernel size of 8 by 8 pixel is used.

- Section 12.4.1.6, 'Edge Sensitive Bayer De-Mosaicing Algorithm'

  Examples - Edge Sensitive Laplace Bayer De-Mosaicing filter

- Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche'

  Examples - Laroche Bayer De-Mosaicing filter

- Section 12.4.1.8, 'Modified Laroche Bayer De-Mosaicing Algorithm '

  Examples - Ressource Optimized Laroche Bayer De-Mosaicing filter

- Section 12.5.2, 'Co-Processor Large Filter Calculation'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a large filter kernel is calculated.

- Section 12.7.2, 'Noise Reduction'

  Examples - The average of two acquired images is calculated to reduce noise.

- Section 12.8.1.2, 'Kirsch Filter'

Examples - The Kirsch filter is a good edge detection filter for non directional edges.

- Section 12.8.1.4, 'Sobel Gradient X'

  Examples - A Sobel filter in x-direction only.

- Section 12.8.1.5, 'Sobel Multi Gradient'

  Examples - A Sobel filter in all 4 directions.

- Section 12.8.3.1, 'Averaging 3x3'

  Examples - A simple 3x3 box filter.

- Section 12.8.3.2, 'Gaussian Filter 5x5'

  Examples - A Gauss filter using a 5x5 kernel.

- Section 12.8.4.1, 'Filter Basics'

  Examples - Explains the implementation of filters.

- Section 12.8.4.2, 'Parallel Filters'

  Examples - An example of the use of two filters in parallel.

- Section 12.8.4.3, 'Filter Sub Kernels'

  Examples - Shows how to extract a sub kernel from a filter to obtain the original image data. This example performs a simple local adaptive binarization.

- Section 12.8.4.4, 'Filter for Line Scan Cameras'

  Examples - Explains how to implement a filter for line scan cameras.

- Section 12.8.5.1, 'High Boost Sharpening Filter'

  Examples - A high boost Laplace filter for sharpening

- Section 12.8.5.2, 'Laplace Filter 3x3'

  Examples - A 3x3 Laplace filter.

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.12.2, 'Depth From Focus Using Loops'

  Examples - Depth From Focus using Loops

- Section 12.13.4, 'Normalized Cross Correlation'

  Examples-

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

# 25.5. Operator HitOrMiss

**Operator Library: Filter**

The operator HitOrMiss performs a morphological hit-or-miss operation of the binary image, which can be used to identify structures or objects. The input is a kernel-stream (see e.g. *FIRkernelNxM*). The output link gives the calculated value for the center pixel. A structuring element for the hit-or-miss operation is given by a parameterizable matrix. The matrix has the size of the input kernel.

The output is set to '1' for a pixel if the kernel neighborhood of the pixel exactly matches with the pattern defined in the parameter *StructElement*. The output is set to '0' if there are one or more mismatches. The search pattern defined with parameter *StructElement* can be either 0 or 1 to define the match. Moreover, value -1 is allowed, too. It defines a don't care value for ignored parts in the structuring element.

Let's have a look at an example. The following struct element is given.

$$H(i,j) = \begin{bmatrix} 1 & -1 & 0 \\ 1 & -1 & 0 \\ 1 & -1 & 0 \end{bmatrix}$$

We will match it with the following image I

$$I(x,y) = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

If we match the structuring element with the image the output O is

$$O(x,y) = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

To generate the required input kernel use operators such as *FIRkernelNxM*, *LineNeighboursNx1* or *PixelNeighbours1xM*.

## 25.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, data output |

## 25.5.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 25.5.3. Parameters

| StructElement | |
|---|---|
| **Type** | static parameter |
| **Default** | -1 |
| **Range** | {-1, 0, 1} |

This parameter defines the shape of the structuring element.

- A '0' in the struct element has to match with a '0' in the respective kernel input.
- A '1' in the struct element has to match with a '1' in the respective kernel input.
- A '-1' in the struct element marks the position as don't care and will be ignored.

If the input kernel size of the operator is changed, the coefficients will also change. Check the coefficients after changing the input kernel size.

### 25.5.4. Examples of Use

The use of operator HitOrMiss is shown in the following examples:

- Section 12.8.2.2, 'Hit or Miss'

  Examples - The implementation can detect four simple patterns in a binary image. For every match, the output will be set to one.

# 25.6. Operator LineNeighboursNx1

**Operator Library: Filter**

The operator LineNeighboursNx1 creates kernel window of N rows and 1 column. The N pixel are located on the top of each input image pixel.

The operator is comparable to *FIRkernelNxM* but comprises other functionality. The key feature of this operator is that it is of O-type what strongly simplifies synchronizations in the design process. However, the price to pay for the O-type functionality are some restrictions.

The first difference to *FIRkernelNxM* is that the operator does not center the current pixel in the kernel. Suppose a kernel of size 3x1 is defined. At kernel index 0 of the output image O(x, y), the current pixel at position I(x, y) is provided. At kernel index 1 of the output image, the pixel of the input image at input position I(x, y-1) is provided. Thus, at kernel row index n the output image is

$$O(x,y,n) = I(x,y-n)$$

In other words, the operator cannot output pixel of 'future' lines i.e. pixels which have not been processed yet. That's the reason why the operator is of O-type and will not cause line or pixel delays.

Because the operator cannot provide information prior to the current line, there is no mirrored edge handling like in *FIRkernelNxM*. Instead all pixels in the output link which origin is outside image borders are set to the value of parameter *Constant*.

To generate a two dimensional kernel, use this operator together with a successive module of operator *PixelNeighbours1xM* in the image processing pipeline.

## 25.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, kernel output |

## 25.6.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | any❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | 1 | any |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 25.6.3. Parameters

| Constant | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | range of input link I |

This parameter defines the default value for pixel outside the image borders.

The value is always unsigned. If you want to set the parameter to a signed value you need to reinterpret the value as unsigned. For color formats, the value is a combined value for all components.

### 25.6.4. Examples of Use

The use of operator LineNeighboursNx1 is shown in the following examples:

- Section 12.9.7, 'Shear of an Image'

  Example - Line Shear example with linear interpolation.

# 25.7. Operator MAX

**Operator Library: Filter**

The operator MAX identifies and outputs the maximum value of an input kernel-stream. This maximum value is output at O. The operator will not output the kernel index of the maximum value.

## 25.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, data output |

## 25.7.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 25.7.3. Parameters

None

## 25.7.4. Examples of Use

The use of operator MAX is shown in the following examples:

- Section 12.8.1.2, 'Kirsch Filter'

  Examples - The Kirsch filter is a good edge detection filter for non directional edges.

# 25.8. Operator MEDIAN

**Operator Library: Filter**

The operator MEDIAN identifies and outputs the median value of an input kernel-stream. The kernel size is limited to 3x3 and 5x5.

## 25.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, data output |

## 25.8.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | {3, 5} | 1 |
| **Kernel Rows** | as kernel columns | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 25.8.3. Parameters

None

## 25.8.4. Examples of Use

The use of operator MEDIAN is shown in the following examples:

- Section 12.5.1, 'Co-Processor Median Filter'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a median filter is calculated.

- Section 12.8.3.3, 'Median Filter 5x5'

  Examples - Applet applies a 5x5 median filter on the image.

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

# 25.9. Operator MIN

**Operator Library: Filter**

The operator MIN identifies and outputs the minimum value of an input kernel-stream. This minimum value is output at O. The operator will not output the kernel index of the minimum value.

## 25.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, data output |

## 25.9.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 25.9.3. Parameters

None

## 25.9.4. Examples of Use

The use of operator MIN is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

  Examples - Demonstration of how to use the operator

# 25.10. Operator NumberOfHits

**Operator Library: Filter**

The operator NumberOfHits counts the number of matches in the binary image of kernel size N x M in the input link I and in the matching matrix. The result is output at the link O. The matching matrix is defined using parameter *StructElement*. An entry of '1' in the *StructElement* indicates that the pixel must contain the value '1' in the kernel to be included in the counting. A value '0' in the structuring element indicates that the pixel must be value '0' to be included in the counting. An entry '-1' marks don't care kernel pixels. Such pixels will be ignored during counting.

Let's have a look at an example. The following structuring element is given.

$$H(i,j) = \begin{bmatrix} -1 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

We will apply the operator to the following image I

$$I(x,y) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The resulting output O is

$$O(x,y) = \begin{bmatrix} 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 3 & 4 & 4 & 4 & 3 \\ 3 & 4 & 4 & 7 & 4 & 5 \\ 5 & 4 & 6 & 4 & 6 & 4 \\ 4 & 6 & 4 & 4 & 4 & 5 \\ 4 & 3 & 4 & 4 & 4 & 4 \end{bmatrix}$$

Assume that the kernel was generated using operator *FIRkernelNxM* with parameter *EdgeHandling* = EdgeMirrored

## 25.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, data output |

## 25.10.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 1 | auto❶ |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Max. Img Height** | any | as I |

❶ The output bit width is automatically determined from the input kernel size by

$$OutputBitWidth = \lceil log_2(InputKernelColumns \times InputKernelRows + 1) \rceil$$

## 25.10.3. Parameters

| **StructElement** | |
|---|---|
| **Type** | static parameter |
| **Default** | -1 |
| **Range** | {-1, 0, 1} |

This parameter defines the shape of the structuring element.

- A '0' in the struct element has to match with a '0' in the respective kernel input to increment the hit counter.
- A '1' in the struct element has to match with a '1' in the respective kernel input to increment the hit counter.
- A '-1' in the struct element marks the position as don't care and will be ignored.

If the input kernel size of the operator is changed, the coefficients will also change. Check the coefficients after changing the input kernel size.

## 25.10.4. Examples of Use

The use of operator NumberOfHits is shown in the following examples:

- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 25.11. Operator PixelNeighbours1xM

**Operator Library: Filter**

The operator PixelNeighbours1xM creates kernel window of 1 row and M columns. The M pixel are located on the left of each input image pixel.

The operator is comparable to *FIRkernelNxM* but comprises other functionality. The key feature of this operator is that it is of O-type what strongly simplifies synchronizations in the design process. However, the price to pay for the O-type functionality are some restrictions.

The first difference to *FIRkernelNxM* is that the operator does not center the current pixel in the kernel. Suppose a kernel of size 1x3 is defined. At kernel index 0 of the output image O(x, y), the current pixel at position I(x, y) is provided. At kernel index 1 of the output image, the pixel of the input image at input position I(x-1, y) is provided. Thus, at kernel column index m the output image is

$$O(x,y,m) = I(x-m,y)$$

In other words, the operator cannot output 'future' pixel, i.e., pixels which have not been processed yet. That's the reason why the operator is of O-type and will not cause line or pixel delays.

Because the operator does not provide information prior to the current pixel, there is no mirrored edge handling like in *FIRkernelNxM*. Instead all pixels in the output link which origin is outside image borders are set to the value of parameter *Constant*.

To generate a two dimensional kernel, use this operator together with an antecedent module of operator *LineNeighborsNxM* in the image processing pipeline.

Operator PixelNeighbours1xM supports variable line lengths.

## 25.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, kernel output |

## 25.11.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | any❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | any |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 25.11.3. Parameters

| Constant | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | range of input link I |

This parameter defines the default value for pixel outside the image borders.

The value is always unsigned. If you want to set the parameter to a signed value you need to reinterpret the value as unsigned. For color formats, the value is a combined value for all components.

### 25.11.4. Examples of Use

The use of operator PixelNeighbours1xM is shown in the following examples:

- Section 12.9.8, 'Scaling a Line Scan Image'

  Examples - Scaling A Line Scan Image

# 25.12. Operator SORT

**Operator Library: Filter**

The operator SORT sorts all elements of an input kernel-stream depending on their gray values. Input link I and output link O both have a kernel field of of size N rows times M columns (I [N, M] and O [N, M]). This kernel field is sorted by the pixel values. The pixel with the maximum gray value will be found on O [0, 0], the 2nd biggest value on O [0, 1], the minimum gray scale value at O [N - 1, M - 1]. Mind that O [ N / 2, M / 2] will output the median of kernel of arbitrary size.

## 25.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, kernel input |
| **Output Link** | O, data output |

## 25.12.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 25.12.3. Parameters

None

## 25.12.4. Examples of Use

The use of operator SORT is shown in the following examples:

- Section 12.13.2, 'Print Inspection Example- Position Correction and Defect Detection Using Blob Based Template Matching'

  Examples- Geometric Transformation and Defect Detection

# 26. Library Logic




The *Logic* library includes operators for logic operations such as comparisons and Boolean operations.

The following list summarizes all Operators of Library Logic

| Operator Name | | Short Description | available since |
|---|---|---|---|
| AND | AND | Performs a bitwise AND operation. | Version 1.1 |
| CASE | CASE | Multiplexes multiple input links of identical formats. | Version 1.1 |
| ≥ | CMP_AgeB | Checks if input link A is greater or equal to the value at the input link B | Version 1.1 |
| > | CMP_AgtB | Checks if input link A is greater than the value at the input link B | Version 1.1 |
| ≤ | CMP_AleB | Checks if input link A is less or equal to the value at the input link B | Version 1.1 |
| < | CMP_AltB | Checks if input link A is less than the value at the input link B | Version 1.1 |
| == | CMP_Equal | Checks if both input links are equal. | Version 1.1 |
| != | CMP_NotEqual | Checks if both input links are not equal. | Version 1.1 |
| IF | IF | Forwards an input to the output if it's corresponding condition input is true. | Version 1.1 |
| ==n | IS_Equal | Compares if the input link value is equal to parameter "Number". | Version 1.1 |
| ≥n | IS_GreaterEqual | Compares if the input link value is grater or equal than parameter "Number". | Version 1.1 |
| >n | IS_GreaterThan | Compares if the input link value is grater than parameter "Number". | Version 1.1 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
| >n< | IS_InRange | Compares if the input link value is within the interval defined by parameters "From" and "To". | Version 1.1 |
| ≤n | IS_LessEqual | Compares if the input link value is less or equal than parameter "Number". | Version 1.1 |
| <n | IS_LessThan | Compares if the input link value is less than parameter "Number". | Version 1.1 |
| !=n | IS_NotEqual | Compares if the input link value is not equal to parameter "Number". | Version 1.1 |
| NOT | NOT | Performs a bitwise NOT of the input bits. | Version 1.1 |
| OR | OR | Performs a bitwise OR operation. | Version 1.1 |
| XNOR | XNOR | Performs a bitwise XNOR operation. | Version 1.1 |
| XOR | XOR | Performs a bitwise XOR operation. | Version 1.1 |

Table 26.1. Operators of Library Logic

# 26.1. Operator AND

**Operator Library: Logic**

The operator AND performs a bitwise logical AND operation. Each output bit is set to a logical "1" if the corresponding bits at all input links are "1", otherwise the output is "0". The number of input links has to be selected at the instantiation of the module.

Thus the output bit i of N inputs is

$$O[i] = I_0[i] \wedge I_1[i] \wedge ... \wedge I_{N-1}[i]$$

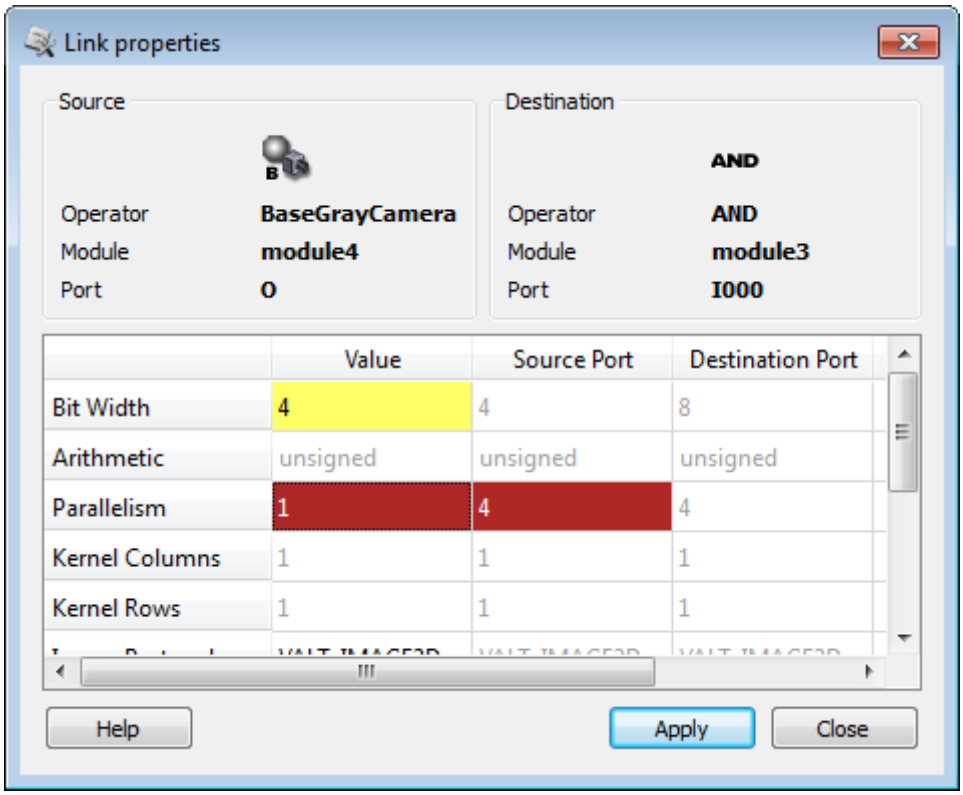
All IN links of the AND operator need to have the same link format as port I[000]. Therefore, you need to configure all IN links accordingly.

You find the link format of port I[000] in the link properties of any IN link connected to operator AND: They are displayed in column "Destination Port", fields "Bit Width" and "Parallelism".

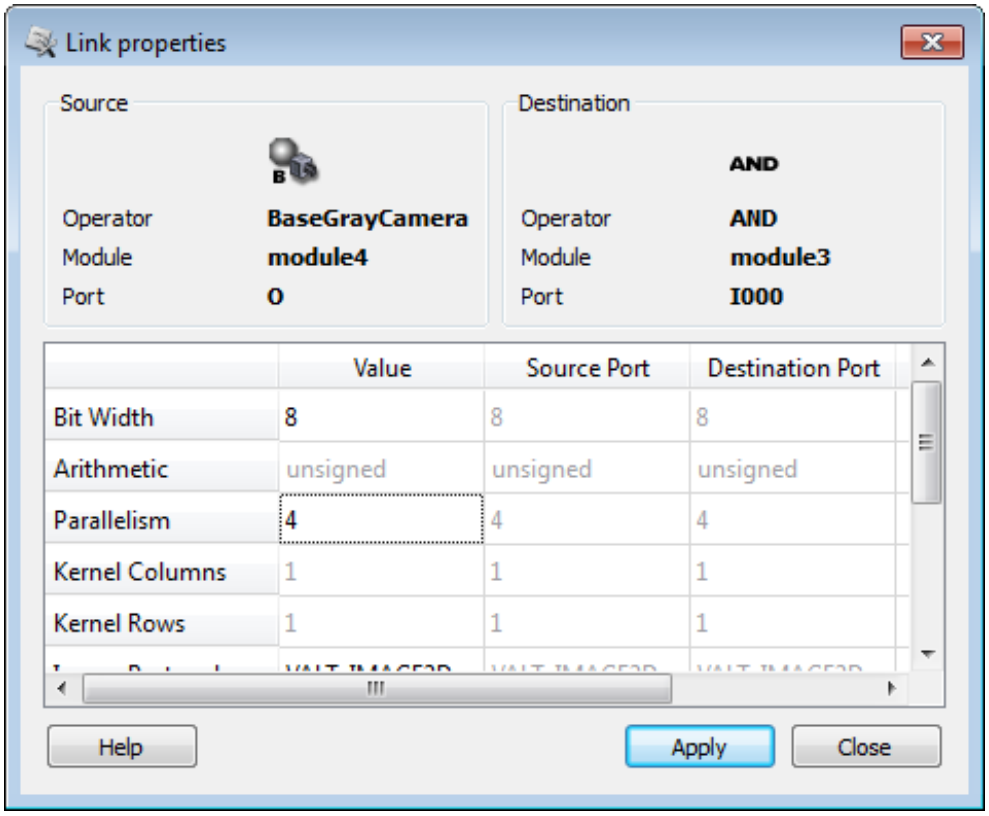
To configure the link format of your IN links to operator AND correctly, you simply enter the values of bit width and parallelism you find in column "Destination Port" into the according fields of column "Value".

The coloring of the link informs you, as always in VisualApplets, about the correctness of your link configuration.

Incorrect configuration of IN link:




Correct configuration of IN link:




### 26.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input<br>I1..IN-1, data input |
| **Output Link** | O, data output |

### 26.1.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1..IN-1 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] | as I | as I |
| **Arithmetic** | {unsigned, signed} | as I | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | as I | as I |
| **Kernel Rows** | any | as I | as I |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D,<br>VALT_SIGNAL} | as I | as I |
| **Color Format** | any | as I | as I |
| **Color Flavor** | any | as I | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 26.1.3. Parameters

None

### 26.1.4. Examples of Use

The use of operator AND is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.3.2, 'Blob 2D'

  Examples - Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels.

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.4.3, 'HSL Color Classification'

  Examples - Color Classification is very simple on HSL images. The applet converts the RGB image into an HSL image and performs a color classification. The hue is filtered using a lookup table. Moreover, the saturation and lightness is thresholded using custom threshold values.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

# 26.2. Operator CASE

**Operator Library: Logic**

The CASE operator multiplexes multiple input links of identical formats. The number of input links has to be selected at the instantiation of the module and determines the bit width of the input link "Switch". The value at input link Switch decides which input link is selected and forwarded to the output. When the Switch input selects a non-existent input link, the output of the operator becomes zero. The operator can be used with kernels. Each kernel component is selected independently.

## 26.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input<br>I1..In-1, data input<br>Switch, data input |
| **Output Link** | O, data output |

## 26.2.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1..In-1 |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I0 |
| **Arithmetic** | {unsigned, signed} | as I0 |
| **Parallelism** | any | as I0 |
| **Kernel Columns** | any | as I0 |
| **Kernel Rows** | any | as I0 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL}❷ | as I0 |
| **Color Format** | any | as I0 |
| **Color Flavor** | any | as I0 |
| **Max. Img Width** | any | as I0 |
| **Max. Img Height** | any | as I0 |

| Link Parameter | Input Link Switch | Output Link O |
|---|---|---|
| **Bit Width** | auto❸ | as I0 |
| **Arithmetic** | unsigned | as I0 |
| **Parallelism** | as I0 | as I0 |
| **Kernel Columns** | as I0 | as I0 |
| **Kernel Rows** | as I0 | as I0 |
| **Img Protocol** | as I0 | as I0 |
| **Color Format** | VAF_GRAY | as I0 |
| **Color Flavor** | FL_NONE | as I0 |
| **Max. Img Width** | as I0 | as I0 |
| **Max. Img Height** | as I0 | as I0 |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ If image protocol VALT_SIGNAL is used, only 2 inputs can be addressed and used.

❸ The bit width of the switch input is automatically determined from the number if input links n

$$SwitchBitWidth = \lceil \log_2(n) \rceil$$

### 26.2.3. Parameters

None

### 26.2.4. Examples of Use

The use of operator CASE is shown in the following examples:

- Section 12.9.7, 'Shear of an Image'

  Example - Line Shear example with linear interpolation.

# 26.3. Operator CMP_AgeB

**Operator Library: Logic**

The operator CMP_AgeB (compare A greater or equal B) sets the output to a logical "1" if the value at the input link A is greater or equal to the value at the input link B, otherwise the output is "0".

To compare the value of input A with a dynamic parameter value use operator *IS_GreaterEqual* instead.

## 26.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | A, data input<br>B, data input |
| **Output Link** | O, data output |

## 26.3.2. Supported Link Format

| Link Parameter | Input Link A | Input Link B | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | as A | unsigned |
| **Parallelism** | any | as A | as I |
| **Kernel Columns** | any | as A | as I |
| **Kernel Rows** | any | as A | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as A | as I |
| **Color Format** | VAF_GRAY | as A | as I |
| **Color Flavor** | FL_NONE | as A | as I |
| **Max. Img Width** | any | as A | as I |
| **Max. Img Height** | any | as A | as I |

## 26.3.3. Parameters

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator.

**EmbeddedALU:** The operator uses embedded ALU elements of the FPGA.

**LUT:** The operator uses the LUT elements of the FPGA.

| ImplementationType |
| --- |
| **Use AUTO in General**<br><br>Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT. |

### 26.3.4. Examples of Use

The use of operator CMP_AgeB is shown in the following examples:

- Section 12.2.1, 'Adaptive Threshold'

  A binarization example for local adaptive thresholding. A kernel size of 8 by 8 pixel is used.

- Section 12.2.2, 'Auto Threshold Mean'

  Determines the mean value of an image and used the value as threshold value for the next image processed.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.8.4.3, 'Filter Sub Kernels'

  Examples - Shows how to extract a sub kernel from a filter to obtain the original image data. This example performs a simple local adaptive binarization.

- Section 13.1, 'Functional Example for Specific Operators of Library Accumulator and Library Logic'

  Examples - Demonstration of how to use the operator

# 26.4. Operator CMP_AgtB

**Operator Library: Logic**

The operator CMP_AgeB (compare A greater than B) sets the output to a logical "1" if the value at the input link A is greater than the value at the input link B, otherwise the output is "0".

To compare the value of input A with a dynamic parameter value use operator *IS_GreaterThan* instead.

## 26.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | A, data input<br>B, data input |
| **Output Link** | O, data output |

## 26.4.2. Supported Link Format

| Link Parameter | Input Link A | Input Link B | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | as A | unsigned |
| **Parallelism** | any | as A | as I |
| **Kernel Columns** | any | as A | as I |
| **Kernel Rows** | any | as A | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as A | as I |
| **Color Format** | VAF_GRAY | as A | as I |
| **Color Flavor** | FL_NONE | as A | as I |
| **Max. Img Width** | any | as A | as I |
| **Max. Img Height** | any | as A | as I |

## 26.4.3. Parameters

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator.

**EmbeddedALU:** The operator uses embedded ALU elements of the FPGA.

**LUT:** The operator uses the LUT elements of the FPGA.

| ImplementationType |
| --- |
| **Use AUTO in General**<br><br>Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT. |

### 26.4.4. Examples of Use

The use of operator CMP_AgtB is shown in the following examples:

- Section 13.1, 'Functional Example for Specific Operators of Library Accumulator and Library Logic'

  Examples - Demonstration of how to use the operator

# 26.5. Operator CMP_AleB

**Operator Library: Logic**

The operator CMP_AleB (compare A less or equal B) sets the output to a logical "1" if the value at the input link A is less or equal to the value at the input link B, otherwise the output is "0".

To compare the value of input A with a dynamic parameter value use operator *IS_LessEqual* instead.

## 26.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | A, data input<br>B, data input |
| **Output Link** | O, data output |

## 26.5.2. Supported Link Format

| Link Parameter | Input Link A | Input Link B | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | as A | unsigned |
| **Parallelism** | any | as A | as I |
| **Kernel Columns** | any | as A | as I |
| **Kernel Rows** | any | as A | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as A | as I |
| **Color Format** | VAF_GRAY | as A | as I |
| **Color Flavor** | FL_NONE | as A | as I |
| **Max. Img Width** | any | as A | as I |
| **Max. Img Height** | any | as A | as I |

## 26.5.3. Parameters

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator.

**EmbeddedALU:** The operator uses embedded ALU elements of the FPGA.

**LUT:** The operator uses the LUT elements of the FPGA.

| ImplementationType |
| --- |
| **Use AUTO in General**<br><br>Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT. |

### 26.5.4. Examples of Use

The use of operator CMP_AleB is shown in the following examples:

- Section 13.1, 'Functional Example for Specific Operators of Library Accumulator and Library Logic'

  Examples - Demonstration of how to use the operator

# 26.6. Operator CMP_AltB

**Operator Library: Logic**

The operator CMP_AltB (compare A less than B) sets the output to a logical "1" if the value at the input link A is less than the value at the input link B, otherwise the output is "0".

To compare the value of input A with a dynamic parameter value use operator *IS_LessThan* instead.

## 26.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | A, data input<br>B, data input |
| **Output Link** | O, data output |

## 26.6.2. Supported Link Format

| Link Parameter | Input Link A | Input Link B | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | as A | unsigned |
| **Parallelism** | any | as A | as I |
| **Kernel Columns** | any | as A | as I |
| **Kernel Rows** | any | as A | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as A | as I |
| **Color Format** | VAF_GRAY | as A | as I |
| **Color Flavor** | FL_NONE | as A | as I |
| **Max. Img Width** | any | as A | as I |
| **Max. Img Height** | any | as A | as I |

## 26.6.3. Parameters

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator.

**EmbeddedALU:** The operator uses embedded ALU elements of the FPGA.

**LUT:** The operator uses the LUT elements of the FPGA.

| ImplementationType |
| --- |
| **Use AUTO in General**<br><br>Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT. |

### 26.6.4. Examples of Use

The use of operator CMP_AltB is shown in the following examples:

- Section 13.1, 'Functional Example for Specific Operators of Library Accumulator and Library Logic'

  Examples - Demonstration of how to use the operator

# 26.7. Operator CMP_Equal

**Operator Library: Logic**

The operator CMP_Equal sets the output to a logical "1" if the value at the input link A is equal to the value at the input link B, otherwise the output is "0".

To compare the value of input A with a dynamic parameter value use operator *IS_Equal* instead.

## 26.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | A, data input<br>B, data input |
| **Output Link** | O, data output |

## 26.7.2. Supported Link Format

| Link Parameter | Input Link A | Input Link B | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | as A | unsigned |
| **Parallelism** | any | as A | as I |
| **Kernel Columns** | any | as A | as I |
| **Kernel Rows** | any | as A | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as A | as I |
| **Color Format** | VAF_GRAY | as A | as I |
| **Color Flavor** | FL_NONE | as A | as I |
| **Max. Img Width** | any | as A | as I |
| **Max. Img Height** | any | as A | as I |

## 26.7.3. Parameters

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator.

**EmbeddedALU:** The operator uses embedded ALU elements of the FPGA.

**LUT:** The operator uses the LUT elements of the FPGA.

| ImplementationType |
| --- |
|  **Use AUTO in General**<br><br>Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT. |

### 26.7.4. Examples of Use

The use of operator CMP_Equal is shown in the following examples:

- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 26.8. Operator CMP_NotEqual

**Operator Library: Logic**

The operator CMP_NotEqual sets the output to a logical "1" if the value at the input link A is not equal to the value at the input link B, otherwise the output is "0".

To compare the value of input A with a dynamic parameter value use operator *IS_NotEqual* instead.

## 26.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | A, data input<br>B, data input |
| **Output Link** | O, data output |

## 26.8.2. Supported Link Format

| Link Parameter | Input Link A | Input Link B | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed | [1, 64] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | as A | unsigned |
| **Parallelism** | any | as A | as I |
| **Kernel Columns** | any | as A | as I |
| **Kernel Rows** | any | as A | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as A | as I |
| **Color Format** | VAF_GRAY | as A | as I |
| **Color Flavor** | FL_NONE | as A | as I |
| **Max. Img Width** | any | as A | as I |
| **Max. Img Height** | any | as A | as I |

## 26.8.3. Parameters

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator.

**EmbeddedALU:** The operator uses embedded ALU elements of the FPGA.

**LUT:** The operator uses the LUT elements of the FPGA.

| ImplementationType |
| --- |
| **Use AUTO in General**<br><br>Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT. |

### 26.8.4. Examples of Use

The use of operator CMP_NotEqual is shown in the following examples:

- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 26.9. Operator IF

**Operator Library: Logic**

The IF operator checks if a condition input is true and forwards the corresponding values of the data input. The operator is equipped with a parameterizable number of input links I0..In-1. With each input link comes a conditional input *Condition*. If a condition is true, the data of its corresponding input is forwarded to the output. If all condition inputs are false, the *Else* input is used. If more than one condition input is true, all corresponding input links are forwarded to the output and combined using a logic OR operation. The operator can be used with kernels. Each kernel component is selected independently.

## 26.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input<br>I1..In-1, data input<br>Else, data input<br>Condition0..n-1, data input |
| **Output Link** | O, data output |

## 26.9.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1..In-1 | Input Link Else |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | as I0 | as I0 |
| **Arithmetic** | {unsigned, signed} | as I0 | as I0 |
| **Parallelism** | any | as I0 | as I0 |
| **Kernel Columns** | any | as I0 | as I0 |
| **Kernel Rows** | any | as I0 | as I0 |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D,<br>VALT_SIGNAL} | as I0 | as I0 |
| **Color Format** | any | as I0 | as I0 |
| **Color Flavor** | any | as I0 | as I0 |
| **Max. Img Width** | any | as I0 | as I0 |
| **Max. Img Height** | any | as I0 | as I0 |

| Link Parameter | Input Link Condition0..n-1 | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I0 |
| **Arithmetic** | unsigned | as I0 |
| **Parallelism** | as I0 | as I0 |
| **Kernel Columns** | as I0 | as I0 |
| **Kernel Rows** | as I0 | as I0 |
| **Img Protocol** | as I0 | as I0 |
| **Color Format** | VAF_GRAY | as I0 |
| **Color Flavor** | FL_NONE | as I0 |
| **Max. Img Width** | as I0 | as I0 |
| **Max. Img Height** | as I0 | as I0 |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 26.9.3. Parameters

None

### 26.9.4. Examples of Use

The use of operator IF is shown in the following examples:

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend'

  Tutorial - From equation to implementation. Explanation on how to implement the overlay blend.

- Section 12.1.6, 'Laser Pointer Detection'

  Examples - A convolution with high intensity spot coefficients is made. For results above threshold, the respective pixels are dyed in red.

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.4.1.1, 'Nearest Neighbor De-Mosaicing'

  Examples - Nearest Neighbor Bayer De-Mosaicing

- Section 12.4.1.6, 'Edge Sensitive Bayer De-Mosaicing Algorithm'

  Examples - Edge Sensitive Laplace Bayer De-Mosaicing filter

- Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche'

  Examples - Laroche Bayer De-Mosaicing filter

- Section 12.4.1.8, 'Modified Laroche Bayer De-Mosaicing Algorithm '

  Examples - Ressource Optimized Laroche Bayer De-Mosaicing filter

- Section 12.4.3, 'HSL Color Classification'

  Examples - Color Classification is very simple on HSL images. The applet converts the RGB image into an HSL image and performs a color classification. The hue is filtered using a lookup table. Moreover, the saturation and lightness is thresholded using custom threshold values.

- Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'

  Examples - High Dynamic Range According to Debevec

- Section 12.10.2, 'High Dynamic Range and Low Dynamic Range Example with a Weighted Linear Ansatz'

  Examples - High Dynamic Range with Linear Ansatz

- Section 12.12.2, 'Depth From Focus Using Loops'

  Examples - Depth From Focus using Loops

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

- Section 12.14.2, 'Grid Overlay Fading'

  Examples - A grid is overlayed to the input images. The grid pixel value is determined from the input pixel value.

# 26.10. Operator IS_Equal

**Operator Library: Logic**

The operator IS_Equal sets the output to a logical "1" if the value at the input link I is equal to the value of parameter *Number*, otherwise the output is "0".

To compare the values of two input links use operator *CMP_Equal* instead.

## 26.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 26.10.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 26.10.3. Parameters

| **Number** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | Same as range of input link I |
| Value to compare the input link value with. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |
| Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.<br><br>You can select one of the following values:<br><br>**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator. | |

| ImplementationType |
| --- |
| **EmbeddedALU:** The operator uses embedded ALU elements of the FPGA.<br><br>**LUT:** The operator uses the LUT elements of the FPGA. |



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

## 26.10.4. Examples of Use

The use of operator IS_Equal is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.14.2, 'Grid Overlay Fading'

  Examples - A grid is overlayed to the input images. The grid pixel value is determined from the input pixel value.

# 26.11. Operator IS_GreaterEqual

**Operator Library: Logic**

The operator IS_GreaterEqual sets the output to a logical "1" if the value at the input link I is greater or equal than the value of parameter *Number*, otherwise the output is "0".

To compare the values of two input links use operator *CMP_AgeB* instead.

## 26.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 26.11.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 26.11.3. Parameters

| **Number** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | same as range of input link I |
| Value to compare the input link value with. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |
| Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.<br><br>You can select one of the following values:<br><br>**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator. | |

| ImplementationType |
| --- |
| **EmbeddedALU:** The operator uses the embedded ALU elements of the FPGA.<br><br>**LUT:** The operator uses the LUT elements of the FPGA. |



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

## 26.11.4. Examples of Use

The use of operator IS_GreaterEqual is shown in the following examples:

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.2.4, 'Simple Threshold Binarization'

  Simple thresholding for binarization.

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.3.2, 'Blob 2D'

  Examples - Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels.

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.4.3, 'HSL Color Classification'

  Examples - Color Classification is very simple on HSL images. The applet converts the RGB image into an HSL image and performs a color classification. The hue is filtered using a lookup table. Moreover, the saturation and lightness is thresholded using custom threshold values.

- Section 12.14.1, 'Dead Pixel Replacement'

  Examples - The examples shows an automatic dead pixel detection and replacement.

# 26.12. Operator IS_GreaterThan

**Operator Library: Logic**

The operator IS_GreaterThan sets the output to a logical "1" if the value at the input link I is greater than the value of parameter *Number*, otherwise the output is "0".

To compare the values of two input links use operator *CMP_AgtB* instead.

## 26.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 26.12.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 26.12.3. Parameters

| **Number** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | same as range of input link I |
| Value to compare the input link value with. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |
| Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.<br><br>You can select one of the following values:<br><br>**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator. | |

| ImplementationType |
| --- |
| **EmbeddedALU:** The operator uses the embedded ALU elements of the FPGA.<br><br>**LUT:** The operator uses the LUT elements of the FPGA. |



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

### 26.12.4. Examples of Use

The use of operator IS_GreaterThan is shown in the following examples:

- Section 9.2, ' Multiple DMA Channel Designs '

  Threshold binarization

- Section 12.8.1.1, 'Morphological Edge'

  Examples - A binary eroded image is compared with the original. An edge is detected if both differ.

- Section 12.8.2.1, 'Close'

  Examples - Shows the implementation of a morphological close applied to binary images.

- Section 12.8.2.3, 'Open'

  Examples - Shows the implementation of a morphological open applied to binary images.

# 26.13. Operator IS_InRange

**Operator Library: Logic**

**Library: Logic**

The operator IS_InRange sets the output to a logical "1" if the value at the input link I is within the interval defined by parameters *From* and *To*, otherwise the output is "0".

$$O = \begin{cases} 1 & \text{if } from \leq I \leq to \\ 0 & \text{otherwise} \end{cases}$$

## 26.13.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 26.13.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 26.13.3. Parameters

| **from** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | Same as range of input link I |
| Smallest value in range. | |

| **to** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 255 |
| **Range** | Same as range of input link I |
| Biggest value in range. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |

| ImplementationType | |
|---|---|
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |

Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator.

**EmbeddedALU:** The operator uses the embedded ALU elements of the FPGA.

**LUT:** The operator uses the LUT elements of the FPGA.



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

## 26.13.4. Examples of Use

The use of operator IS_InRange is shown in the following examples:

- Section 12.13.2, 'Print Inspection Example- Position Correction and Defect Detection Using Blob Based Template Matching'

  Examples- Geometric Transformation and Defect Detection

# 26.14. Operator IS_LessEqual

**Operator Library: Logic**

The operator IS_LessEqual sets the output to a logical "1" if the value at the input link I is less or equal than the value of parameter *Number*, otherwise the output is "0".

To compare the values of two input links use operator *CMP_AleB* instead.

## 26.14.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 26.14.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 26.14.3. Parameters

| **Number** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | same as range of input link I |
| Value to compare the input link value with. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |
| Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.<br><br>You can select one of the following values:<br><br>**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator. | |

| ImplementationType |
| --- |
| **EmbeddedALU:** The operator uses the embedded ALU elements of the FPGA.<br><br>**LUT:** The operator uses the LUT elements of the FPGA. |



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

## 26.14.4. Examples of Use

The use of operator IS_LessEqual is shown in the following examples:

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend'

  Tutorial - From equation to implementation. Explanation on how to implement the overlay blend.

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.3.2, 'Blob 2D'

  Examples - Shows the usage of operator *Blob_Analysis_2D*. The applet binarizes the input data and determines the blob analysis results. The results as well as the original image are output using two DMA channels.

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.4.3, 'HSL Color Classification'

  Examples - Color Classification is very simple on HSL images. The applet converts the RGB image into an HSL image and performs a color classification. The hue is filtered using a lookup table. Moreover, the saturation and lightness is thresholded using custom threshold values.

# 26.15. Operator IS_LessThan

**Operator Library: Logic**

The operator IS_LessThan sets the output to a logical "1" if the value at the input link I is less than the value of parameter *Number*, otherwise the output is "0".

To compare the values of two input links use operator *CMP_AltB* instead.

## 26.15.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 26.15.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 26.15.3. Parameters

| **Number** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | same as range of input link I |
| Value to compare the input link value with. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |
| Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.<br><br>You can select one of the following values:<br><br>**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator. | |

| ImplementationType |
| --- |
| **EmbeddedALU:** The operator uses the embedded ALU elements of the FPGA.<br><br>**LUT:** The operator uses the LUT elements of the FPGA. |



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

## 26.15.4. Examples of Use

The use of operator IS_LessThan is shown in the following examples:

- Section 12.1.6, 'Laser Pointer Detection'

  Examples - A convolution with high intensity spot coefficients is made. For results above threshold, the respective pixels are dyed in red.

# 26.16. Operator IS_NotEqual

**Operator Library: Logic**

The operator IS_NotEqual sets the output to a logical "1" if the value at the input link I is not equal to the value of parameter *Number*, otherwise the output is "0".

To compare the values of two input links use operator *CMP_NotEqual* instead.

## 26.16.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 26.16.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 63] unsigned, [2, 64] signed | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 26.16.3. Parameters

| **Number** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | same as range of input link I |
| Value to compare the input link value with. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, EmbeddedALU, LUT) |
| Parameter ImplementationType allows you to influence the implementation of the operator, i.e., to define which logic elements are used for implementing the operator.<br><br>You can select one of the following values:<br><br>**AUTO:** When the operator is instantiated, the optimal implementation strategy for the given FPGA architecture is selected automatically, based on the parametrization of the operator. | |

| ImplementationType |
| --- |
| **EmbeddedALU:** The operator uses the embedded ALU elements of the FPGA.<br><br>**LUT:** The operator uses the LUT elements of the FPGA. |



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values EmbeddedALU and/or LUT.

## 26.16.4. Examples of Use

The use of operator IS_NotEqual is shown in the following examples:

- Section 9.2, ' Multiple DMA Channel Designs '

  Remove 9 out of 10 images.

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

# 26.17. Operator NOT

**Operator Library: Logic**

The operator NOT inverts the input bits. It performs a bitwise logical inversion. Each output bit is set to a logical "1" if the corresponding input bit is "0", otherwise the output is "0". The number of input links has to be selected at the instantiation of the module.

Thus the output bit i is

$O[i] = \overline{I[i]}$

## 26.17.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 26.17.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 26.17.3. Parameters

None

## 26.17.4. Examples of Use

The use of operator NOT is shown in the following examples:

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend'

  Tutorial - From equation to implementation. Explanation on how to implement the overlay blend.

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

# 26.18. Operator OR

**Operator Library: Logic**

The operator OR performs a bitwise logical OR operation. Each output bit is set to a logical "1" if any of the corresponding bits is "1", otherwise the output is "0". The number of input links has to be selected at the instantiation of the module.

Thus the output bit i of n inputs is

$$O[i] = I_0[i] \vee I_1[i] \vee ... \vee I_{n-1}[i]$$

## 26.18.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input<br>I1..In-1, data input |
| **Output Link** | O, data output |

## 26.18.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1..In-1 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | as I | as I |
| **Arithmetic** | {unsigned, signed} | as I | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | as I | as I |
| **Kernel Rows** | any | as I | as I |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D,<br>VALT_SIGNAL} | as I | as I |
| **Color Format** | any | as I | as I |
| **Color Flavor** | any | as I | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 26.18.3. Parameters

None

## 26.18.4. Examples of Use

The use of operator OR is shown in the following examples:

- Section 12.8.2.2, 'Hit or Miss'

  Examples - The implementation can detect four simple patterns in a binary image. For every match, the output will be set to one.

- Section 12.14.2, 'Grid Overlay Fading'

Examples - A grid is overlayed to the input images. The grid pixel value is determined from the input pixel value.

# 26.19. Operator XNOR

**Operator Library: Logic**

The operator XNOR performs a bitwise logical XNOR (not exclusive or) operation. Each output bit is set to a logical "1" if the sum of the corresponding input bit values is even, otherwise the output is "0". The number of input links has to be selected at the instantiation of the module.

Thus the output bit i of n inputs is

$$O[i] = \overline{(I_0[i] \text{ XOR } I_1[i] \text{ XOR } ... \text{ XOR } I_{n-1}[i])}$$

## 26.19.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input<br>I1..In-1, data input |
| **Output Link** | O, data output |

## 26.19.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1..In-1 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | as I | as I |
| **Arithmetic** | {unsigned, signed} | as I | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | as I | as I |
| **Kernel Rows** | any | as I | as I |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D,<br>VALT_SIGNAL} | as I | as I |
| **Color Format** | any | as I | as I |
| **Color Flavor** | any | as I | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 26.19.3. Parameters

None

## 26.19.4. Examples of Use

The use of operator XNOR is shown in the following examples:

- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 26.20. Operator XOR

**Operator Library: Logic**

The operator XOR performs a bitwise logical XOR (exclusive OR) operation. Each output bit is set to a logical "1" if the sum of the corresponding input bit values is odd, otherwise the output is "0". The number of input links has to be selected at the instantiation of the module.

Thus the output bit i of n inputs is

$$O[i] = I_0[i] \text{ XOR } I_1[i] \text{ XOR } ... \text{ XOR } I_{n-1}[i]$$

## 26.20.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I0, data input<br>I1..In-1, data input |
| **Output Link** | O, data output |

## 26.20.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1..In-1 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | as I | as I |
| **Arithmetic** | {unsigned, signed} | as I | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | as I | as I |
| **Kernel Rows** | any | as I | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I | as I |
| **Color Format** | any | as I | as I |
| **Color Flavor** | any | as I | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 26.20.3. Parameters

None

## 26.20.4. Examples of Use

The use of operator XOR is shown in the following examples:

- Section 12.8.1.1, 'Morphological Edge'

  Examples - A binary eroded image is compared with the original. An edge is detected if both differ.

# 27. Library Memory




The *Memory* library includes operators for buffering data, resorting data, and random access to data. Many different operators exist. They all implement their own idea of memory access.

The operators are either using *frame grabber RAM (DRAM)*, FPGA-internal block RAM (*BRAM* or *URAM*), or *FPGA distributed RAM (LUT RAM)*. The following table shows an overview of which memory type is used by the individual operators.

| Name | Memory Type |
|---|---|
| *CoefficientBuffer* | Frame grabber RAM (DRAM) |
| *FrameBufferMultiRoiDyn* | Frame grabber RAM (DRAM) |
| *FrameBufferRandomRead* | Frame grabber RAM (DRAM) |
| *FrameMemory* | FPGA-internal block RAM |
| *FrameMemoryRandomRd* | FPGA-internal block RAM |
| *ImageBuffer* | Frame grabber RAM (DRAM) |
| *ImageBufferMultiRoI* | Frame grabber RAM (DRAM) |
| *ImageBufferMultiRoIDyn* | Frame grabber RAM (DRAM) |
| *ImageBufferSC* | Frame grabber RAM (DRAM) |
| *ImageBufferSpatial* | Frame grabber RAM (DRAM) |
| *ImageFifo* | FPGA-internal RAM |
| *ImageSequence* | Frame grabber RAM (DRAM) |
| *KneeLUT* | FPGA-internal block RAM |
| *LineBuffer* | Frame grabber RAM (DRAM) |
| *LineMemory* | FPGA-internal RAM |
| *LineMemoryRandomRd* | FPGA-internal block RAM |
| *LUT* | FPGA-internal block RAM |
| *RamLUT* | Frame grabber RAM (DRAM) |
| *ROM* | FPGA-internal block RAM |

Table 27.1. Memory Types of Operators in the Library Memory

Operators might require additional FPGA-internal RAM for buffering even if they are using *frame grabber RAM (DRAM)*.

The delay, i.e., latency of a memory operator, depends on the operator's implementation. Some operators only have a short pixel delay, while others have a line or frame delay. The following table lists the delay of the individual memory operators.

| Name | Latency |
|---|---|
| *CoefficientBuffer* | None: Will always output data if output is not blocked. |
| *FrameBufferMultiRoiDyn* | Minimum 1 frame: Reading using coordinate inputs starts after frame has been fully written into the buffer. If output is blocked, operator buffers input data. |
| *FrameBufferRandomRead* | Minimum 1 frame: As soon as frame is fully written reading starts using address inputs. If output is blocked, operator buffers input data. Latency also depends on presence of read address data. |

| Name | Latency |
|---|---|
| *FrameMemory* | 1 frame: Reading starts after frame has been fully written into the buffer. |
| *FrameMemoryRandomRd* | 1 frame: Reading can start after frame has been fully written into the buffer. Latency depends on presence of read address data. |
| *ImageBuffer* | Minimum 1 line: Output starts reading a line as soon as it is fully written into the buffer. If output is blocked, operator buffers input data. |
| *ImageBufferMultiRoI* | Minimum 1 frame: Reading starts after frame has been fully written into the buffer. If output is blocked, operator buffers input data. |
| *ImageBufferMultiRoIDyn* | Minimum 1 frame: Reading using coordinate inputs starts after frame has been fully written into the buffer. If output is blocked, operator buffers input data. |
| *ImageBufferSC* | Minimum 1 line: Output starts reading a line as soon as it is fully written into the buffer. If output is blocked, operator buffers input data. |
| *ImageBufferSpatial* | Minimum 1 line: Output starts reading a line as soon as it is fully written into the buffer. If output is blocked, operator buffers input data. |
| *ImageFifo* | No latency. Only if output is blocked, operator buffers and delays input data. |
| *ImageSequence* | Minimum SequenceLength frames: Reading starts after all frames of a sequence have been fully written into the buffer. If output is blocked, operator buffers input data. |
| *KneeLUT* | No latency |
| *LineBuffer* | 1 line: Reading starts after line has been fully written into the buffer. |
| *LineMemory* | 1 line: Reading starts after line has been fully written into the buffer. |
| *LineMemoryRandomRd* | 1 line: Reading can start after line has been fully written into the buffer. Latency depends on presence of read address data. |
| *LUT* | No latency |
| *RamLUT* | Latency of some clock cycles only. Depends on addresses. |
| *ROM* | No Latency |

Table 27.2. Individual Latencies of the Operators in Library Memory

The following list summarizes all Operators of Library Memory

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | CoefficientBuffer | Allows the upload of image files from PC to frame grabber and uses them as image source. | Version 1.1 |
|  | FrameBufferMultiRoiDyn | Buffers the input images in the *Frame Grabber RAM (DRAM)* and reads out multiple dynamic regions of interest (ROI) for each buffered image. | Version 3.5 |
|  | FrameBufferRandomRead | FrameBufferRandomRead is a memory buffer with random read access. | Version 1.3 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | FrameBufferRandomRead (imaFlex) | FrameBufferRandomRead (imaFlex) is a memory buffer for the imaFlex CXP-12 Quad and the imaFlex CXP-12 Penta platforms with random read access. | Version 3.3 |
|  | FrameMemory | FrameMemory is a small memory block with random write access. | Version 1.1 |
|  | FrameMemoryRandomRd | FrameMemoryRandomRd is a small memory block with random read access. | Version 1.3 |
|  | ImageBuffer | Buffers an image in *Frame Grabber RAM (DRAM)* with ROI selection. | Version 1.1 |
|  | ImageBufferMultiRoI | Buffers the input image stream in *Frame Grabber RAM (DRAM)* and outputs an arbitrary number of ROIs from each input image. | Version 1.1 |
|  | ImageBufferMultiRoIDyn | Buffers the input images in *Frame Grabber RAM (DRAM)* and transfers an arbitrary number of dynamic ROIs out of each image. | Version 1.1 |
|  | ImageBufferSC | Buffers an image in *Frame Grabber RAM (DRAM)* with sensor correction and ROI selection. | Version 1.1 |
|  | ImageBufferSpatial | Buffers the image in the *Frame Grabber RAM (DRAM)* and computes a spatial correction of lines. | Version 1.3 |
|  | ImageFifo | Buffers a limited number of pixels in the FPGA internal RAM. | Version 1.1 |
|  | ImageSequence | Buffers a sequence of images and output them simultaneously. | Version 1.1 |
|  | KneeLUT | Implements an approximation of a Lookup table by a series of nodes. | Version 1.1 |
|  | LineBuffer (imaFlex) | Buffers image data line-by-line in the *Frame Grabber RAM (DRAM)* with region-of-interest (ROI) support. This operator is only available for the imaFlex CXP-12 Quad and the imaFlex CXP-12 Penta platform. | Version 3.3 |
|  | LineMemory | LineMemory is a small memory block with random write access. | Version 1.1 |
|  | LineMemoryRandomRd | LineMemoryRandomRd is a small memory block with random read access. | Version 1.3 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | LUT | Implements a Lookup table of dynamic content. | Version 1.1 |
|  | RamLUT | Implements a large Lookup table of dynamic content based on FPGA external RAM (i.e., the *Frame Grabber RAM (DRAM)*). | Version 1.3 |
|  | RamLUT (imaFlex) | Implements a large lookup table of dynamic content based on FPGA-external RAM. | Version 3.3 |
|  | ROM | Implements a Lookup table of dynamic content. | Version 1.1 |

Table 27.3. Operators of Library Memory

# 27.1. Operator CoefficientBuffer

**Operator Library: Memory**

The operator allows the upload of image files from the host PC to the frame grabber which will be continuously output and used as image source. An image file is used here as a universal data container. Therefore, uploaded images can contain any data such as coefficients. All coefficient images must have the same pixel width and height but can be in completely different formats. For example, link 0 coefficient image can be in GRAY 8 bit format. Link 1 could provide an image in RGB16 format. And link 2 could be a binary image. All coefficient images are stored in the *Frame Grabber RAM (DRAM)*. One VisualApplets resource of type *RAM* is required. Check Section 3.8, 'Allocation of Device Resources' for more information. For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.



***CoefficientBuffer* Is Not Available for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta Platforms**

*CoefficientBuffer* is not supported on imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms. The example Section 11.4, 'Functional Example for Loading Test Images Using *ImageInjector* ' shows how you can substitute the functionality of the *CoefficientBuffer* operator as test image source on imaFlex CXP-12 Quad or imaFlex CXP-12 Penta using the *ImageInjector* operator. Note that for the *ImageInjector* operator you need either an **Expert** license, a **Debugging Module** license, or the **VisualApplets 4** license.

The example Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT' shows how you can a *RamLUT* instead of the *CoefficientBuffer* for 2D shading correction.

The number of links corresponds to the number of coefficient images stored in CoefficientBuffer. The coefficient images can only be uploaded into the FPGA buffer via software while no image acquisition is running. This is done by writing to parameter *LoadCoefficients*.

The output ports of the operator work synchronously. The transfer of the first image pixel starts exactly at the same moment on all output ports. If all images have been transferred to the receiving operator, CoefficientBuffer starts to synchronously output the images again.



**Behavior during simulation**

The images on the operator's links are output simultaneously. In one simulation step, the operator outputs maximally one image on each link. If during a simulation step all of the stored images (one per link) have been forwarded to the receiving operator, i.e., if there occurred no blocking, the same output (one image per link) will be forwarded in the following simulation step.

CoefficientBuffer also supports a region of interests (RoI) management. The RoI is defined by the 4 parameters XOffset, XLength, YOffset and YLength. XOffset and YOffset define the upper left corner of the RoI, i.e., the start coordinate in pixels. The dimensions of the RoI must not exceed the dimensions of the stored coefficient images defined by the parameters *BufferWidth* and *BufferHeight*. The RoI can be updated dynamically during acquisition or, when no acquisition is running, by setting all 4 RoI coordinate parameters to new values and writing to the parameter UpdateROI. Note that XLength must also fulfill the granularity of the link parallelism, i.e., XLength modulo LinkParallelism = 0.

For each output link, a coefficient image file is stored in the buffer, i.e., coefficient image 0 will be provided on link 0. The coefficient images must be loaded into CoefficientBuffer before the acquisition starts. The coefficient images must be stored in the files specified by the parameters CoefficientFile N. N stands for the link number respectively for the coefficient image number. The coefficient files must be encoded in TIF format.

*BufferWidth* specifies the width of all coefficient images stored in CoefficientBuffer in pixels.

*BufferHeight* specifies the height of all coefficient images managed by CoefficientBuffer in pixels. *BufferWidth* and *BufferHeight* determine the output maximal image dimensions of all output links.

For grayscale links, grayscale image files have to be used. For color links, RGB image files have to be used.

If the bit width of a link is higher than the bit width of the image file, the operator will use multiple image file pixel to generate the output pixels. For example, a link of 22 bit will require 3 pixel in an 8 bit image file. Thus, the image width of the image file has to be 3 times higher than parameter *BufferWidth*.

If the bit width of a link is less than the bit width of the image file, the operator will use the lower bits of the input image file only. For example, an 8 bit image file and a 3 bit link will use bits 0, 1 and 2 of the file.

16 bit TIF files should only be used with care.

Parameter LoadCoefficients is used to start loading of coefficient images into the buffer before the image acquisition starts. The loading is triggered by a write cycle or value 1 to this parameter. Writing value 0 has no effect.

Parameter UpdateROI is used to update the current RoI dynamically while the image acquisition is running or when no images are grabbed. The new RoI must be specified by the 4 RoI coordinate parameters before writing to this parameter. The written value does not affect the updating process, only the write cycle triggers it, i.e., the written value can be either 0 or 1.

Parameter XOffset specifies the offset in horizontal direction from the left image border to the 1st column of the RoI. The specified value is measured in pixels and must not exceed BufferWidth-1 value.

Parameter XLength specifies the length of the RoI in horizontal direction in pixels. The specified value must not exceed BufferWidth and must be divisible by the output link parallelism. Also the following constraint must be met: XOffset + XLength <= BufferWidth.

Parameter YOffset specifies the offset in vertical direction from the top image border to the 1st line of the RoI. YOffset must be specified in pixels and must not exceed BufferHeight-1.

Parameter YLength specifies the length of the RoI in vertical direction in pixels. The length must not exceed BufferHeight. Furthermore, the following constraint must be met: YOffset + YLength <= BufferHeight.

Parameter CoefficientFile N specifies the file name of the coefficient image for link N stored in TIF format.



**Note**

1. The parallelism of link 0 determines the parallelism of all other links.
2. Setting of the parallelism for the link 0 corrects XLength value to be divisible by the new parallelism. This might be handled differently in future VA versions. This means that when using the up/down buttons in the spinbox of the parallelism attribute of the link XLength might be modified. Therefore a check on XLength setting is recommended after all links are connected and specified.
3. Setting the parameter BufferWidth and BufferHeight corrects the RoI coordinates in case any of them exceeds the new image dimensions. The coordinates will be corrected to the closest possible valid values.
4. CoefficientBuffer supports 2D, 1D and 0D images. In case of 1D and 0D images the operator will provide the coefficient images - that are stored as 2D images - continuously without generating end of frame respectively end of line markers.
5. The operator is using 1 DDRRAM bank. Thus, the storage space for coefficient images is limited to the capacity of one RAM bank (e.g., 128MByte on mE4 boards, 256MByte on mE5 boards) per 1 operator. Furthermore, due to the internal formatting of the DDRRAM, the actual storage space might be a bit less than the physical storage space. The used storage space is related to the link formats respectively to the pixel width of each link. In an optimal case the sum of all link pixel widths is divisible by the capacity of one RAM bank (i.e., by 128 on me4, or by 256 on mE5). In this case all

coefficient images can be stored together in one memory bank. In any other case the storage will be used sub-optimally. Check Appendix A, '*Device Resources*' for more information on hardware platforms.

Recommended Setting Order for using this Operator

1. Connect all links of CoefficientBuffer.
2. Specify the link formats for all links.
3. Set BufferWidth and BufferHeight parameters.
4. Set RoI coordinates.
5. Set the coefficient file names.

### 27.1.1. Using the Operator with Maximum Performance

Internally, the CoefficientBuffer operator generates one cumulative pixel from all coefficient image pixels. This is the so-called super-pixel.

- If the RAM data width is less than the super-pixel bit width, the super-pixel is distributed over multiple RAM cells.
- If the super pixel is less than the RAM data width, only one super-pixel is stored in each RAM cell, even if multiple super-pixels would fit into a RAM cell.

Therefore, a RAM cell will only be occupied by one super-pixel or is part of a super-pixel.

The operator internally stores a pixel of the output bit width in a RAM-data-width wide block, e.g., if the output bit width is set to 8bit on a marathon VCL platform (RAM data width is 256bit), only 8 bit of each RAM cell are used. The rest is wasted.

Increasing the bandwidth by simply increasing the parallelism will **NOT** help in this case.

You can look up the RAM data width of the frame grabber you are using in the Platform Resources section of this documentation:

Appendix A, '*Device Resources*'

Find below some ideas to increase the bandwidth/ RAM-efficiency:

Tip 1:

To increase the bandwidth, you may define the bitwidth of a pixel to be bigger, and later on use a CastParallel operator to correct this.

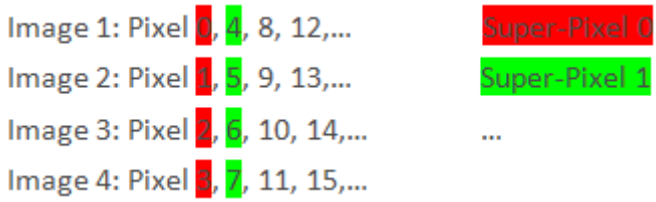
Example: You need an image in Gray 8bit but with the bandwidth of parallelism 8. For that you can define the output bit width to be 64 (parallelism 1) and later on use operator CastParallel to shift the link back to 8 bit and parallelism 8.

This way, your image will also consume less RAM as more than 1 pixel are stored in 1 RAM cell. If used directly - without operator CastParallel - the memory needed to store the image will be 8 times bigger.

Tip 2:

If the bit width of the combined link would exceed 64 bit, you can split your image to multiple images and use more than one link of operator CoefficientBuffer. Later on you can combine all links since all outputs are O-synchronous.

Example: You want to send an RGB24 image in parallelism 4. The accumulated pixel would be 4*24 = 96Bit - which is too big. So the idea to solve this would be to split the image into 4 sub images, each having ¼ of the pixel.




Internally, the operator will accumulate the pixel of each input to one super-pixel of 96bit.

This way, the efficiency of the ram is also increased. If we take a 1024x1024 image on an Ironman mE5VQ8-CXPD platform (128 Bit wide RAM data interface), the image stored in simple 1 link CoefficientBuffer is

1024 * 1024 * 128 / 8 Byte = 16MB.

If the 4-link setup is applied, the size is

1024 * 1024 * (128 / 4) / 8 = 4MB

#### 27.1.1.1. Example for microEnable IV frame grabbers with a RAM cell width of 64 bit.

Suppose that the Coefficientbuffer is used with a single coeffcient image file only, the pixel bit width is 8 bit per pixel and the parallelism is 1. In this case, the buffer will dramatically loose performance. That's because only 1/8th of the bandwidth is used. Moreover, the buffer reads its its values with doubled design clock frequency, i.e., only 1/16th of the full bandwidth is used.

To use the buffer with maximum performance, the pixel data has to be reinterpreted. In out example, the buffer output link has to be parameterized to 64 bit per pixel and parallelism two has to be used. This will cause the buffer to read 128 bit per design clock. Use a CastParallel operator to reinterpret the 64 bit data values to a bit width of eight and a parallelism of 16. Moreover, the parameter *BufferWidth* has to be set to 1/8th of the image width of the coefficient image.

In conclusion, the maximum bandwidth can only be obtained if the output bit width is high, e.g., 64 bit. Using a high output parallelism will not increase the bandwidth.

### 27.1.2. Using the Operator as Simulation Source

The operator can be used as a simulation source. In this case, the operator will try to load and use the image files specified by the *Coefficient File N* parameters. Note that this will only work properly if the image files exactly match with the link format. If you have a link format of e.g. 64 Bit, the image width of your image file has to be 8 times the maximum image width of the link as the image will have 8 bit per pixel. The height has to exactly match with the maximum image height specified at the link, too.

### 27.1.3. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width x 2 (true for read-only parameters).



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

### 27.1.4. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O[n], data output |

### 27.1.5. Supported Link Format

| Link Parameter | Output Link O[n] |
|---|---|
| **Bit Width** | [1, 64]❶❷ |
| **Arithmetic** | {unsigned, signed} |
| **Parallelism** | any❷ |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} |
| **Color Format** | any |
| **Color Flavor** | any |
| **Max. Img Width** | BufferWidth |
| **Max. Img Height** | BufferHeight |

❶ The range of the output bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The product of the bit width and the parallelism must not exceed the native ram data width. Check Appendix A, '*Device Resources*' for more information.

### 27.1.6. Parameters

| **RamDataWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of data bits that can be used at the RAM interface. | |

| **RamAddressWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of RAM address bits available. | |

| **BufferWidth** | |
|---|---|
| **Type** | static parameter |
| **Default** | 1024 |
| **Range** | [1, 65535] |
| This parameter defines coefficient images width in pixels. Moreover, this parameter defines the maximum image width of all output links. The range of this parameter depends on the used frame grabber and its RAM. | |

| **BufferHeight** | |
|---|---|
| **Type** | static parameter |

| BufferHeight | |
|---|---|
| **Default** | 1024 |
| **Range** | [1, 65535] |
| This parameter defines the coefficient images height in pixels. Moreover, this parameter defines the maximum image height of all output links. The range of this parameter depends on the used frame grabber and its RAM. | |

| LoadCoefficients | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | {0, 1} |
| This parameter is used to start loading of coefficient images into the buffer before the image acquisition starts. The loading is triggered by a write cycle of value one to this parameter. Writing value 0 does not cause the loading of the coefficient files. | |

| UpdateROI | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1 |
| **Range** | {1} |
| The parameter is used to update the ROI settings in hardware. The values defined by the 4 ROI coordinate parameters are used. An update can be done at any time, even during a running acquisition. | |

| XOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Width - XLength] |
| This parameter defines the x-coordinate of the upper left corner of the ROI. Changing the value does not directly change the ROI settings. You have to write to parameter *UpdateROI* to apply new settings.<br><br>The step size is the parallelism. | |

| XLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [parallelism, Max.Img Width - XOffset] |
| This parameter defines the width of the ROI. Changing the value does not directly change the ROI settings. You have to write to parameter *UpdateROI* to apply new settings.<br><br>The step size is the parallelism. | |

| YOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, BufferHeight - YLength] |
| This parameter defines the y-coordinate of the upper left corner of the ROI. Changing the value does not directly change the ROI settings. You have to write to parameter *UpdateROI* to apply new settings. | |

| YLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |

| YLength | |
|---|---|
| **Default** | 1024 |
| **Range** | [1, BufferHeight - YOffset] |
| This parameter defines the height of the ROI. Changing the value does not directly change the ROI settings. You have to write to parameter *UpdateROI* to apply new settings. | |

| FillLevel | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of DRAM in 25% steps. | |

| Overflow | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| This parameter indicates a buffer overflow. | |

| Coefficient File N | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | coefficients_N.tif |
| **Range** | any filename of a tif file with optional path |
| This parameter defines the file name of the coefficient image for the link N. | |

### 27.1.7. Examples of Use

The use of operator CoefficientBuffer is shown in the following examples:

- Section 12.6.4, 'Manual Image Injection'

  Example - For debugging purposes images can be inserted manually.

- Section 12.6.5, 'Image Monitoring'

  Example - For debugging purposes image transfer states on links can be investigated.

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

## 27.2. Operator FrameBufferMultiRoiDyn

**Operator Library: Memory**

This operator buffers input image data in the *frame grabber RAM (DRAM)* and reads out multiple dynamic regions of interest (ROI) for each buffered image. The coordinates of the ROIs to be read out are defined by using additional input links. The following image shows an overview of the operator's behavior for one input frame and four ROIs.

One VisualApplets resource of type *RAM* is required (see Section 3.8, 'Allocation of Device Resources'). Multiple resources of type *RAM* use the same physical RAM with the Section A.3, ' Shared Memory Concept '. Documentation for how to use the shared memory is available in the Application Note: Shared Memory [https://docs.baslerweb.com/visualapplets/application-note-shared-memory].

All ROIs are read sequentially as individual images: ROI 0, ROI 1, ROI 2,.. until ROI N-1. N is the number of ROIs to be read out for one input frame, that is the image size on the coordinate inputs. All possible rectangular regions are supported as long as the maximum input frame dimensions are not exceeded, i.e. a single pixel, a single line, a single column, a rectangular region, or the complete frame can be defined as an ROI.

Different ROIs do not need to have the same size.

Each ROI can be defined individually.

ROIs can overlap each other.

If the ROI image is empty, i.e. it doesn't contain any pixels, the operator provides an empty image on its output. An empty image contains no pixels.

A negative ROI width (*XTopLeft* > *XBottomRight*) or ROI height (*YTopLeft* > *YBottomRight*) also leads to an empty ROI on the output. The maximum amount of supported ROIs is determined by the maximum number of pixels per image on the *XTopLeft* input link: *XTopLeft.MaxImageWidth* * *XTopLeft.MaxImageHeight* .

ROI coordinates are provided on four separate coordinate input links as images containing the ROI coordinates: *XTopLeft*, *YTopLeft*, *XBottomRight*, *YBottomRight*. Each pixel in these images is treated by the operator as a valid ROI coordinate set. Thus, the size of the ROI input images define the number of output ROIs. **EoL**s (End of Lines) and varying line lengths can be used but are ignored as only valid pixels and **EoF**s (End of Frames) are evaluated for the ROI inputs of the operator. For each input image at input link I, a new ROI coordinate image set is required. The ROIs are defined by the X,Y coordinates for the top left corner and the X,Y coordinates for the bottom right corner.

Note that ROI X-coordinates are transformed to meet the X-granularity of the input link I defined by its parallelism, i.e., the operator *FrameBufferMultiRoiDyn* can only cut lines with the granularity of the parallelism. When addressing a pixel that isn't the first pixel of a parallel component for *XTopLeft*, or a pixel that isn't the last pixel of a parallel component for *XBottomRight*, the coordinates are rounded up or down and the resulting ROI still contains all pixels of the parallel component.

The following section describes an example to explain the operator's behavior:

In this example, the parallelism = 4 at the input link I.

Therefore, *XTopLeft* is rounded down to the first pixel of a parallel component: 0, 4, 8, 12, ... and *XBottomRight* is rounded up to the last pixel of a parallel component 3, 7, 11, 15, ...

For example, with the ROI X-coordinates 3 and 7, the first column of the ROI is column number 3 and the last column is column number 7 of the input image. The operator rounds down *XTopLeft* to 0; *XBottomRight* remains 7. Thus, the width becomes 8. The ROI height is not dependent on the parallelism and thus can be of any legal value.



**Formula:**

XTopLeft* = floor(XTopLeft / Parallelism) * Parallelism

XBottomRight* = ceil((XBottomRight + 1) / Parallelism) * Parallelism - 1



**Examples:**

With a parallelism of 4 at the input link I:

- 112 to 551 becomes 112 to 551. Therefore, the width becomes = 551 - 112 + 1 = 440
- 101 to 540 becomes 100 to 543. Therefore, the width becomes = 543 - 100 + 1 = 444

All 4 ROI inputs are synchronous to each other, i.e., the images on these links must be of the same size and provided at the same time synchronously. This is always the case when they are sourced by the same M type module through an arbitrary network of O-type modules. For more information, see the section Section 3.6.4, 'M-type Operators with Multiple Inputs'.

The ROI coordinates are asynchronous to the input link I. You can use the *CreateBlankImage* operator to generate ROI coordinates.

The operator provides ROI images at its output as soon as the correspondent input frame is completely buffered, i.e. the input frame's **EoF** (End of Frame) signal is received, and the first pixel on each of the ROI image definition input links *XTopLeft, YTopLeft, XBottomRight, YBottomRight* is received. If enough space is available in the DRAM, the next input frames can be written to the DRAM while ROIs are read. The RAM bandwidth is shared between writing and reading but reading can only start after the corresponding input frame is written. After reading the last ROI for the input frame, which is defined by the last pixel on each of the ROI image definition input links (i.e. the ROI images **EoF** (End of Frame) is received), the current frame is discarded and the next ROI images address the next input frame. For information about the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The operator supports two modes for buffering image data to the DRAM, which you can configure via the parameter *MaxFrameSizeMode*:

- For *MaxFrameSizeMode* = *AUTO*, the DRAM buffers an image that has the size of the maximum image dimension, which is *I.MaxImgWidth* * *I.MaxImgHeight*.

  Images that are smaller than the maximum image dimension are buffered without modifications. However, reading out ROIs is always done on a buffered frame with the maximum possible image dimension. In this case, the missing frame pixels are filled with random data and should be treated in further processing as undefined.

- For *MaxFrameSizeMode* = *CUSTOM*, the maximum dimensions of the buffered input frame are defined by the parameters *MaxFrameWidth* and *MaxFrameHeight*. Even if the actual input frame is bigger, the data that exceeds the specified parameter dimensions is discarded (similar to an ROI without offsets). In that case, the DRAM reserves memory for the frame size *MaxFrameWidth* * *MaxFrameHeight*.

In both cases, there is no check whether the input link ROI coordinates are located in the input image. As the coordinates can be set to any pixel of the maximum input frame. If they are located outside of the actual input image data, the operator reads random dummy values, but the ROI size is not altered.

The operator supports empty frames, empty lines and varying line lengths on all input links.

### 27.2.1. Overflow Management with InfiniteSource

In the *InfiniteSource* mode, images might get lost or become corrupted. This happens either, because the RAM is full and it can't accept any further data, or the input bandwidth is too high for the shared memory interface. As soon as the operator reaches the overflow state, all incoming data is discarded. This leads to lost or corrupted frames. Corrupted images only occur, if the input bandwidth is too high, since the fill level is counted in full entities. A corrupted image occurs when part of the image has already been written to the RAM, but part of the image is discarded because the input bandwidth is too high and the RAM can't accept the write data fast enough. As a result, reading a corrupted image

leads to undefined output data. As soon as there is enough space in the RAM again and the shared memory interface allows data to be written to the RAM again, the operator recovers from the overflow and stops discarding the input data. The *Overflow* parameter indicates when an overflow occurred. With the *OverflowClearMode* parameter, you can define whether the *Overflow* is reset immediately after overflow recovery or whether you reset the overflow manually.

## 27.2.2. Operator Restrictions

- The parallelism of the input link must be a power of 2.
- The operator supports empty frames on input link I but the output ROIs for that frame are filled with random dummy values.
- If the input image is smaller than the frame that is written to DRAM, there is no check whether the ROI coordinates are inside the input image. In this case, the operator reads random dummy values from the memory.
- The maximum link properties differ depending on the usage of the operator. Refer to the section *Supported Link Format* below for more information.

## 27.2.3. Bandwidth Optimization

For optimal performance, the used number of data bits should match as closely as possible the number provided in the module parameter *RamDataWidth*. The maximum bandwidth going through the operator is reached, if the product of bit width, kernel size and parallelism is equal to the internal RAM port width *RamDataWidth*. Note that the internal bit width can be increased by the usage of kernels, but a full kernel will still be addressed as a single pixel.

$$DataWidth_{optimal} = BitWidth \cdot KernelColumns \cdot KernelRows \cdot Parallelism = RamDataWidth$$

## 27.2.4. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image data input<br>XTopLeft, coordinate data input<br>YTopLeft, coordinate data input<br>XBottomRight, coordinate data input<br>YBottomRight, coordinate data input |
| **Output Link** | O, data input |

**Synchronous and Asynchronous Inputs**

- The 4 ROI coordinate inputs *XTopLeft*, *YTopLeft*, *XBottomRight* and *YBottomRight* are synchronous to each other.
- The ROI coordinate inputs *XTopLeft*, *YTopLeft*, *XBottomRight* and *YBottomRight* are asynchronous to input link *I*.

## 27.2.5. Supported Link Format

| Link Parameter | Input Link I | Input Link XTopLeft | Input Link YTopLeft |
|---|---|---|---|
| **Bit Width** | [1, 64]❶❷ | auto❸ | auto❹ |
| **Arithmetic** | {unsigned, signed} | unsigned | unsigned |
| **Parallelism** | 2^N, with N = {0,1,2...}❷ | 1 | 1 |

| Link Parameter | Input Link I | Input Link XTopLeft | Input Link YTopLeft |
|---|---|---|---|
| **Kernel Columns** | [1, RamDataWidth / BitWidth / Parallelism / KernelRows]❷ | 1 | 1 |
| **Kernel Rows** | [1, RamDataWidth / BitWidth / Parallelism / KernelColumns]❷ | 1 | 1 |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | VALT_IMAGE2D |
| **Color Format** | any | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | any | FL_NONE | FL_NONE |
| **Max. Img Width** | 2^31 - 1❺ | 2^31 - 1 | as XTopLeft |
| **Max. Img Height** | 2^RamAddressWidth❺ | 2^31 - 1 | as XTopLeft |

| Link Parameter | Input Link XBottomRight | Input Link YBottomRight | Output Link O |
|---|---|---|---|
| **Bit Width** | auto❸ | auto❹ | as I |
| **Arithmetic** | unsigned | unsigned | as I |
| **Parallelism** | 1 | 1 | as I |
| **Kernel Columns** | 1 | 1 | as I |
| **Kernel Rows** | 1 | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | as I |
| **Color Format** | VAF_GRAY | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | FL_NONE | as I |
| **Max. Img Width** | as XTopLeft | as XTopLeft | as I |
| **Max. Img Height** | as XTopLeft | as XTopLeft | as I |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

The input bit width must not exceed the native RAM data width *RamDataWidth*.

❷ The product of bit width, parallelism and kernel size must not exceed the native RAM data width.

$$BitWidth \cdot KernelColumns \cdot KernelRows \cdot Parallelism \leq RamDataWidth$$

❸ The bit width of the X-coordinate inputs is:

$$BitWidth = \lceil log_2(MaxImageWidth(I) - 1) \rceil$$

❹ The bit width of the Y-coordinate inputs is:

$$BitWidth = \lceil log_2(MaxImageHeight(I) - 1) \rceil$$

❺ The maximum image width and image height of the input link *I* differ depending on the operator use:

- For *MaxFrameSizeMode* = *AUTO*, the product of *I.MaxImageWidth* and *I.MaxImageHeight* must fit into the available RAM size.

$$\frac{I.MaxImgWidth}{I.Parallelism} \cdot I.MaxImgHeight \leq 2^{RamAddressWidth}$$

- For *MaxFrameSizeMode* = *CUSTOM*, both *I.MaxImageWidth* and *I.MaxImageHeight* are limited to the maximum RAM size. The product can exceed the total RAM size as only the sub-image selected by the parameters *MaxFrameWidth* and *MaxFrameHeight* is written into the RAM.

$$I.MaxImgHeight \leq 2^{RamAddressWidth}$$

As *I.Parallelism* pixels are written per RAM vector, *I.MaxImageWidth* can exceed 2^RamAddressWidth but it can never be greater than 2^31 - 1.

$$\frac{I.MaxImgWidth}{I.Parallelism} \leq 2^{RamAddressWidth}$$

The sum of the bit widths of *I.MaxImageWidth* and *I.MaxImageHeight* must be smaller than 48 bit.

$$log_2(I.MaxImageWidth) + log_2(I.MaxImageHeight) < 48$$

### 27.2.6. Parameters

| **RamDataWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the maximum data width that can be used in the RAM. | |

| **RamAddressWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of address bits that can be used. The number of available RAM slots is defined by 2^*RamAddressWidth*. The current *RamAddressWidth* depends on the hardware platform as well as on the current number of memory operators (i.e. operators that use a resource of type RAM) in the design. | |

| **MaxFrameSizeMode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Auto |
| **Range** | {Auto, Custom} |
| This parameter defines whether the parameters *MaxFrameWidth* and *MaxFrameHeight* should automatically follow the max. image dimension at input *I* (mode=*Auto*) or whether these parameters can be adjusted (mode=*Custom*). *Custom* mode enables setting a maximum frame dimension, which is customized for the application and may be much smaller than the limits defined by the input link. This way, more frames can be stored in the RAM. In mode *Auto*, the link properties (*Max.Img Width* and *Max.Img Height*) of *I*, must not define a maximum image that is greater than the number of available slots in the memory (2^*RamAddressWidth*). If the link properties *Max.Img Width* and *Max.Img Height* of *I* define a maximum image that is bigger than the available memory space and you are in mode *Custom*, the *MaxFrameSizeMode* parameter can't be edited anymore until the maximum image on the input link is defined smaller. This parameter can't be written when the acquisition is running. | |

| **MaxFrameWidth** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [1, Max. Image Width at I] |
| This parameter sets the max. image width for the current image processing configuration. The lines of input frames that exceed this limit are cut to *MaxFrameWidth*. Reducing this number below the max. image width saves memory space and allows storing more frames. This parameter can only be edited when *MaxFrameSizeMode* is set to *Custom*. The product of | |

| **MaxFrameWidth** |
| --- |
| *MaxFrameHeight* and *MaxFrameWidth* must not be greater than the number of available memory slots 2^*RamAddressWidth* multiplied with the input parallelism. This parameter can't be edited when the acquisition is running. |

| **MaxFrameHeight** | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [1, Max. Image Height at I] |
| This parameter sets the max. image height for the current image processing configuration. Input frames exceeding this height limit are cut to *MaxFrameHeight*. Reducing this number below the max. image height saves memory space and allows to store more frames. This parameter can only be edited when *MaxFrameSizeMode* is set to *Custom*. The product of *MaxFrameHeight* and *MaxFrameWidth* must not be greater than the number of available memory slots 2^*RamAddressWidth* multiplied with the input parallelism. This parameter cannot be edited when the acquisition is running. | |

| **FillLevel** | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of the DRAM in percent. | |

| **MaxFrameCount** | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 2^*RamAddressWidth* / (*MaxFrameHeight* * (*MaxFrameWidth* / *I.Parallelism*)) |
| **Range** | [1, 2^*RamAddressWidth*] |
| This parameter provides the maximum number of frames that currently fit into the memory. The maximum number of frames that fit into the memory depends on the parameters *RamAddressWidth*, *MaxFrameHeight*, *I.Parallelism* and *MaxFrameWidth*.<br><br>MaxFrameCount = 2^*RamAddressWidth* / (*MaxFrameHeight* * (*MaxFrameWidth* / *I.Parallelism*)) | |

| **FrameCount** | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, *MaxFrameCount*] |
| This parameter provides the current number of frames in the memory. | |

| **InfiniteSource** | |
| --- | --- |
| **Type** | static write parameter |
| **Default** | DISABLED |
| **Range** | {ENABLED, DISABLED} |
| The operator can be placed directly behind a camera operator in the design. In this case, the *InfiniteSource* parameter must be set to *ENABLED*. The operator will then perform active overflow management and make sure the operator can properly recover from overflows. The overflow can occur either when the data sink behind the operator stops or pauses the transmission and the buffer fill level reaches its maximum or when the input bandwidth is too high so the write data can't be transferred to the external RAM. When *InfiniteSource* is set to *DISABLED*, an inhibit signal is generated that stops the proceeding operator from transferring data, if the buffer fill level or input bandwidth get too high.<br><br>See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

| **Overflow** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 3] |

This parameter indicates a buffer overflow. It's a 2-bit bitmap, where each bit indicates a different type of overflow. Bit 0 indicates a fill level overflow and bit 1 indicates a write bandwidth overflow. The display time of an *Overflow* depends on the selected *OverflowClearMode*.

| **OverflowClearMode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | AutoClear |
| **Range** | {AutoClear, ManualClear, ClearAfterRead, ClearWithProcessReset} |

*OverflowClearMode* determines how the *Overflow* parameter is cleared when the operator has recovered from an overflow. You can only reset the overflow status with this parameter, if the operator is not in overflow state anymore.

Clear modes:

- *AutoClear*: When the operator recovers from an overflow, the *Overflow* parameter is reset automatically.
- *ManualClear*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until it is manually reset by writing *ManualClear* into the *OverflowClearMode* parameter. In this mode, a process reset (e.g. acquisition stop) doesn't clear the *Overflow* parameter, which means the overflow is still visible after the acquisition stopped.
- *ClearAfterRead*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until the *Overflow* parameter is read or a process reset occurs (e.g. when the acquisition is stopped).
- *ClearWithProcessReset*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until a reset occurs (e.g. when the acquisition is stopped).

## 27.3. Operator FrameBufferRandomRead

**Operator Library: Memory**

This operator facilitates the random read of the buffered data. The frame data is transferred into the operator via link I. The random read of the data can be performed by transferring addresses to ports *RColA* (read column address) and *RRowA* (read row address). The operator will use the addresses to read the frame data by the given coordinates. The resulting output frame will have the image dimensions of the address inputs. One VisualApplets resource of type *RAM* is required. Check Section 3.8, 'Allocation of Device Resources' for more information. For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The FrameBufferRandomRead (frame buffer random read) operator buffers the image stream in the *Frame Grabber RAM (DRAM)*. One VisualApplets resource of type *RAM* is required.

The operator has two states:

1. Write State: Input images are stored in the buffer. The operator works like a FIFO. If enough memory is available, multiple images can be stored in the buffer.
2. Read State: After an image has been fully stored in the memory, the memory is read-out using the addresses given at the inputs RColA and RRowA. The number of addresses and address link image dimensions define the image output width and height.

The image data input and the address inputs are not synchronous to each other. They may have different image dimensions. Both address inputs RColA and RRowA have to be O-synchronous.

Please note the timing of the input links. The address inputs must not be sourced by the same operator as the data link input I without buffering. This is because while writing the image data into the operator, no addresses to read the current frame can be accepted. Only if the frame is fully stored into the buffer, addresses can be accepted. In many cases, the operator *CreateBlankImage* is used to generate the images for the addresses.

The operator uses large non-FPGA memory. If only small frames or lines have to be stored consider using operators *FrameMemoryRandomRd* or *LineMemoryRandomRd*.

To measure the fill level of the buffer the operator provides 2 parameters: *FillLevel* and *Overflow*. FillLevel shows the percentage fill level of the RAM. The Overflow parameter is set to 1 when FillLevel is close to or is 100% and the next image to be stored in the buffer will exceed the RAM capacity. In case of an overflow, input frames are discarded. Users have to poll for the overflow parameter. As the duration of the overflow state can be very short it is possible that it is in between of a polling cycle of the operator.

Parameter *InfiniteSource* is used to specify if the operator is directly connected to a camera or is sequenced with other memory operators. Check Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information.

**Operator Restrictions**

- Empty frames are not supported.
- Images with varying line lengths are not supported.
- The parallelism on the address ports has to be set to 2.
- *I.MaxImgWidth* * *I.MaxImgHeight* must not exceed the available RAM size: 2^*RamAddressWidth*.

### 27.3.1. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width.



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

## 27.3.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image data input<br>RColA, read column address for the pixel at I<br>RRowA, read row address for the pixel at I |
| **Output Link** | O, data input |

**Synchronous and Asynchronous Inputs**

- Synchronous Group: *RColA* and *RRowA*
- Input *I* is asynchronous to the group.

## 27.3.3. Supported Link Format

| Link Parameter | Input Link I | Input Link RColA |
|---|---|---|
| **Bit Width** | [1, 64]❶ | auto❷ |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | 1 | {1, 2}❸ |
| **Kernel Columns** | [1, Ram Data Width / Bit Width / Kernel Rows]❹ | 1 |
| **Kernel Rows** | [1, Ram Data Width / Bit Width / Kernel Columns]❹ | 1 |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any❺ | any |
| **Max. Img Height** | any❺ | any |

| Link Parameter | Input Link RRowA | Output Link O |
|---|---|---|
| **Bit Width** | auto❻ | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | as RColA | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | as RColA | as RColA |
| **Max. Img Height** | as RColA | as RColA |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

The input bit width must not exceed the native ram data width. Check Appendix A, '*Device Resources*' for more information.

❷ The bit width of the column address is:

$$ColABitWidth = \lceil log_2(InputMax.ImageWidth) \rceil$$

❻ The bit width of the row address is:

$$RowABitWidth = \lceil log_2(InputMax.ImageHeight) \rceil$$

❸ The parallelism on the address ports has to be set to 2 in order to use the operator in an efficient way.

❹ *I.BitWidth* x *I.Kernel Columns* x *I.Kernel Rows* <= *RamDataWidth*.

❺ *I.MaxImgWidth* * *I.MaxImgHeight* must not exceed the available RAM size: 2^*RamAddressWidth*.

### 27.3.4. Parameters

| **RamDataWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of data bits that can be used at the RAM interface. It's the maximum number of bits for input and output. | |

| **RamAddressWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of address bits available. This helps to calculate the maximum allowed image dimensions and the maximum address width. | |

| **FillLevel** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of RAM. | |

| **Overflow** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| This parameter indicates a buffer overflow. | |

| **InfiniteSource** | |
|---|---|
| **Type** | static parameter |
| **Default** | ENABLED |

| InfiniteSource | |
|---|---|
| **Range** | {ENABLED, DISABLED} |
| This parameter activates support for infinite source operators like Camera operators. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

## 27.3.5. Examples of Use

The use of operator FrameBufferRandomRead is shown in the following examples:

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.9.3.2.1, 'Geometric Transformation Using **FrameBufferRandomRead**'

  Examples- Geometric Transformation using **FrameBufferRandomRead**

- Section 12.9.3.2.4, 'Geometric Transformation and Distortion Correction'

  Examples- Geometric Transformation and Distortion Correction using **PixelReplicator**

- Section 12.9.3.2.5, 'Distortion Correction'

  Examples- Distortion Correction

- Section 12.9.8, 'Scaling a Line Scan Image'

  Examples - Scaling A Line Scan Image

- Section 12.9.9, 'Tap Geometry Sorting'

  Examples - Scaling A Line Scan Image

- Section 12.10.3, 'Image Composition Using Exposure Fusion'

  Examples - ExposureFusion

# 27.4. Operator FrameBufferRandomRead (imaFlex)

**Operator Library: Memory**

This operator facilitates the random read of the buffered data. The frame data is transferred into the operator via link I. The random read of the data can be performed by transferring addresses to ports *RColA* (read column address) and *RRowA* (read row address). The operator uses the addresses to read the frame data by the given coordinates. The resulting output frame has the image dimensions of the address inputs. One VisualApplets resource of type *RAM* is required. Check Section 3.8, 'Allocation of Device Resources' for more information.

The *FrameBufferRandomRead* operator buffers the image stream in the Frame Grabber RAM (DRAM). One VisualApplets resource of type *RAM* is required. Multiple resources of type *RAM* use the same physical RAM with the shared memory concept. Documentation for how to use the shared memory is available in the Application Note: Shared Memory [https://docs.baslerweb.com/visualapplets/application-note-shared-memory].

The operator has two states:

- *Write* state: Input images are stored in the buffer. The operator works like a FIFO. If enough memory is available, multiple images can be stored in the buffer.
- *Read* state: After an image has been fully stored in the memory, the memory is read-out using the addresses given at the inputs *RColA* and *RRowA*. The number of addresses and address link image dimensions define the output image's width and height.

The image data input and the address inputs are not synchronous to each other. They may have different image dimensions. Both address inputs *RColA* and *RRowA* have to be O-synchronous.

Note the timing of the input links: The address inputs must not be sourced by the same operator as the data link input I without buffering. This is, because while writing the image data into the operator, no addresses to read the current frame can be accepted. Only if the frame is fully stored into the buffer, addresses can be accepted. In many cases, the operator *CreateBlankImage* is used to generate the images for the addresses.

The operator uses large non-FPGA memory. If only small frames or lines have to be stored, consider using operators *FrameMemoryRandomRd* or *LineMemoryRandomRd*.

The operator can handle both, a two-dimensional image (*Img Protocol* = *VALT_IMAGE2D*), as well as a one-dimensional sequence of lines (*Img Protocol* = *VALT_LINE1D*). In the two-dimensional mode, full frames are processed, which means that reading can only start when a full frame has been written. Once the address ports (*RColA* and *RRowA*) receive the end of a frame, the frame's memory is available again and the next read addresses the next frame. The one-dimensional mode processes lines. As a result, reading can start as soon as one line has been written. The end of a line on the address port *RColA* releases the line's memory and the next *RColA* value will address the next line. *RRowA* is not used in the one-dimensional line mode and should be set to any constant value (*CONST*).

All pins allow empty frames, empty lines and varying line lengths.

The dimensions of input frames may exceed the dimension of the internal frames. The operator then cuts the incoming frames to the dimensions defined by *MaxFrameWidth* and *MaxFrameHeight*.

The maximum dimensions of the input images on port *I* can be defined bigger than there is actual storage in the RAM available. This allows for a flexible usage that can switch between very wide and very high images. The parameters *MaxFrameWidth* and *MaxFrameHeight* define the size of the image in storage. As a result, those two parameters must not define an image that is greater than the available storage.

## 27.4.1. Overflow Management with InfiniteSource

In the *InfiniteSource* mode images might get lost or become corrupted. This happens either, because the RAM is full and it can't accept any further data. Or the input bandwidth is too high for the shared memory interface. As soon as the operator reaches the overflow state, all incoming data is discarded. This leads to lost or corrupted entities (i.e. frames in 2D mode or lines in 1D mode). Corrupted images

only occur, if the input bandwidth is too high, since the fill level is counted in full entities. The corrupted image occurs when part of the image has already been written to the RAM, but part of the image is discarded due to the overflow. As a result, reading a corrupted image leads to undefined output data. As soon as there is enough space in the RAM again and the shared memory interface allows data to be written to the RAM again, the operator recovers from the overflow and stops discarding the input data. The *Overflow* parameter indicates when an overflow occurred. With the *OverflowClearMode* parameter, you can define whether the *Overflow* is reset immediately after overflow recovery or whether you reset the overflow manually.

## 27.4.2. Operator Restrictions

- There is no validity check on the read addresses. If the *RColA* is greater than or equal to *MaxFrameWidth* or if *RRowA* is greater than or equal to *MaxFrameHeight* (only in 2D mode), the operator reads random dummy values from the memory.
- The restrictions for the link properties are always a maximum of 2^24-1 for the *Max. Image Width* and 2^17-1 for the *Max. Image Height*. However, if *MaxFrameSizeMode* is set to *Auto*, the following restrictions apply additionally:
  - In 2D-mode, if *MaxFrameSizeMode* is set to *Auto*, then *I.MaxImgWidth* * *I.MaxImgHeight* <= 2^*RamAddressWidth*.
  - In 1D-mode, if *MaxFrameSizeMode* is set to *Auto*, then *I.MaxImgWidth* <= 2^*RamAddressWidth*.
- The parallelism of all input ports is 1.

## 27.4.3. Bandwidth Optimization

For optimal performance the used number of data bits should match as closely as possible the number provided in the module parameter *RamDataWidth*. The maximum bandwidth going through the operator is reached, if the product of bit width and kernel size is equal to the internal RAM port width *RamDataWidth*. Note that the internal bit width can be increased by the usage of kernels, but a full kernel will still be addressed as a single pixel.

| Available for Hardware Platform |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |

## 27.4.4. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image data input<br>RColA, read column address for the pixel at I<br>RRowA, read row address for the pixel at I |
| **Output Link** | O, image data output |

**Synchronous and Asynchronous Inputs**

- Synchronous Group: *RColA* and *RRowA*
- Input *I* is asynchronous to the group.

## 27.4.5. Supported Link Format

| Link Parameter | Input Link I | Input Link RColA |
|---|---|---|
| **Bit Width** | [1, 64]❶ | auto❷ |

| Link Parameter | Input Link I | Input Link RColA |
|---|---|---|
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | [1, Ram Data Width / Bit Width / Kernel Rows]❸ | 1 |
| **Kernel Rows** | [1, Ram Data Width / Bit Width / Kernel Columns]❹ | 1 |
| **Img Protocol** | VALT_LINE1D / VALT_IMAGE2D | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | 2^24-1❺ | any |
| **Max. Img Height** | 2^17-1❺ | any |

| Link Parameter | Input Link RRowA | Output Link O |
|---|---|---|
| **Bit Width** | auto❻ | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | as RColA | as RColA |
| **Max. Img Height** | as RColA | as RColA |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

The input bit width must not exceed the native RAM data width *RamDataWidth*.

❷ The bit width of the column address is:

$$ColABitWidth = \lceil log_2(InputMax.ImageWidth) \rceil$$

❻ The bit width of the row address is:

$$RowABitWidth = \lceil log_2(InputMax.ImageHeight) \rceil$$

❸ BitWidth * Columns * Rows <= *RamDataWidth*

❹ BitWidth * Columns * Rows <= *RamDataWidth*

❺ The restrictions for the link properties are always a maximum of 2^24-1 for the *Max. Image Width* and 2^17-1 for the *Max. Image Height*. However, if *MaxFrameSizeMode* is set to *Auto*, the following restrictions apply additionally:

- In 2D-mode, if *MaxFrameSizeMode* is set to *Auto*, then *I.MaxImgWidth* * *I.MaxImgHeight* <= 2^*RamAddressWidth*.
- In 1D-mode, if *MaxFrameSizeMode* is set to *Auto*, then *I.MaxImgWidth* <= 2^*RamAddressWidth*.

### 27.4.6. Parameters

| RamDataWidth | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of data bits that can be used. | |

| RamAddressWidth | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of address bits that can be used. The number of available RAM slots is defined by 2^*RamAddressWidth*. The current *RamAddressWidth* depends on the hardware platform as well as on the current number of memory operators (i.e. operators that use a resource of type RAM) in the design. | |

| MaxFrameSizeMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Auto |
| **Range** | {Auto, Custom} |
| This parameter defines whether the parameters *MaxFrameWidth* and *MaxFrameHeight* should automatically follow the max. image dimension at input *I* (mode=*Auto*) or whether these parameters can be adjusted (mode=*Custom*). *Custom* mode enables setting a maximum frame dimension, which is customized for the application and may be much smaller than the limits from the input link. This way, more frames can be stored in the RAM. In mode *Auto* the link properties (*Max.Img Width* and *Max.Img Height*) of *I*, must not define a maximum image that is greater than the number of available slots in the memory (2^*RamAddressWidth*). If the link properties *Max.Img Width* and *Max.Img Height* of *I* define a maximum image that is bigger than the available memory space and you are in mode *Custom*, the *MaxFrameSizeMode* parameter cannot be edited anymore until the maximum image on the input link is defined smaller. This parameter cannot be written when the acquisition is running. In 1D-mode *MaxFrameHeight* and *Max.Img Height* of input *I* can be considered to be 1. | |

| MaxFrameWidth | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [1, Max. Image Width at I] |
| This parameter sets the max. image width for the current image processing configuration.<br>The lines of input frames that exceed this limit are cut to *MaxFrameWidth*. Reducing this number below the max. image width saves memory space and allows storing more frames.<br>This parameter can only be edited when *MaxFrameSizeMode* is set to *Custom*. The product of *MaxFrameHeight* and *MaxFrameWidth* must not be greater than the number of available memory slots: 2^*RamAddressWidth*. In 1D-mode *MaxFrameHeight* can be considered to be 1. This parameter cannot be edited when the acquisition is running. | |

| MaxFrameHeight | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1024 |
| **Range** | [1, Max. Image Height at I] |
| This parameter sets the max. image height for the current image processing configuration. Input frames exceeding this height limit are cut to *MaxFrameHeight*. Reducing this number below the max. image height saves memory space and allows to store more frames. This parameter | |

| **MaxFrameHeight** |
|---|
| can only be edited when *MaxFrameSizeMode* is set to *Custom* and the *Image Protocol* is set to *VALT_IMAGE2D*. In 1D-mode *MaxFrameHeight* can be considered to be 1. The product of *MaxFrameHeight* and *MaxFrameWidth* must not be greater than the number of available memory slots: 2^*RamAddressWidth*. This parameter cannot be edited when the acquisition is running. |

| **FillLevel** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of the DRAM in percent. | |

| **MaxFrameCount** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 2^*RamAddressWidth* / (*MaxFrameHeight* * *MaxFrameWidth*) |
| **Range** | [1, 2^*RamAddressWidth*] |
| This parameter provides the maximum number of frames that currently fit into the memory. The maximum number of frames that fit into the memory depends on the parameters *RamAddressWidth*, *MaxFrameHeight* and *MaxFrameWidth*.<br><br>MaxFrameCount = 2^*RamAddressWidth* / (*MaxFrameHeight* * *MaxFrameWidth*)<br><br>This parameter is inactive in 1D mode. | |

| **FrameCount** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, *MaxFrameCount*] |
| This parameter provides the current number of frames in the memory.<br><br>This parameter is inactive in 1D mode. | |

| **InfiniteSource** | |
|---|---|
| **Type** | static write parameter |
| **Default** | DISABLED |
| **Range** | {ENABLED, DISABLED} |
| The operator can be inserted directly behind a camera operator. In this case, the *InfiniteSource* parameter must be set to *ENABLED*. The operator will then perform active overflow management and make sure the operator can properly recover from overflows. The overflow can occur either when the data sink behind the operator stops or pauses the transmission and the buffer fill level reaches its maximum or when the input bandwidth is too high so the write data can't be transferred to the external RAM. When *InfiniteSource* is set to *DISABLED*, an inhibit signal is generated that stops the proceeding operator from transferring data, if the buffer fill level or input bandwidth get too high.<br><br>See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

| **Overflow** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 3] |
| This parameter indicates a buffer overflow. It's a 2-bit bitmap, where each bit indicates a different type of overflow. Bit 0 indicates a fill level overflow and bit 1 indicates a write bandwidth overflow. How long the *Overflow* parameter shows an overflow, depends on the *OverflowClearMode*. | |

| OverflowClearMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | AutoClear |
| **Range** | {AutoClear, ManualClear, ClearAfterRead, ClearWithProcessReset} |

*OverflowClearMode* determines how the *Overflow* parameter is cleared when the operator has recovered from an overflow. You can only reset the overflow status with this parameter, if the operator is not in overflow state anymore.

Clear modes:

- *AutoClear*: When the operator recovers from an overflow, the *Overflow* parameter is reset automatically.
- *ManualClear*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until it is manually reset by writing *ManualClear* into the *OverflowClearMode* parameter. In this mode, a process reset (e.g. acquisition stop) doesn't clear the *Overflow* parameter, which means the overflow is still visible after the acquisition stopped.
- *ClearAfterRead*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until the *Overflow* parameter is read or a process reset occurs (e.g. when the acquisition is stopped).
- *ClearWithProcessReset*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until a reset occurs (e.g. when the acquisition is stopped).

# 27.5. Operator FrameMemory

**Operator Library: Memory**

The FrameMemory operator is a memory block which stores input images using random write access. The write addresses are specified with input links row address RowA and column address ColA. Thus, for each input pixel value, a row and column address has to be specified.

After a frame has been written to the memory, it is read and output using link O. The output image dimension is defined with parameters and is independent of the input image dimension. The memory is pre-initialized with values zero.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The memory size is defined by parameters *FrameWidth* and *FrameHeight*.

The operator can be implemented in two variants. Namely, a single buffer implementation and a double buffer implementation. The first implementation saves RAM, but does not allow a timely overlap of writing and reading. Thus the input link I is stopped during the read state. The latter implementation doubles the RAM and allows to output a frame while accepting the next frame at the input link I.

The operator has two states:

1. Write State: The value of Link I is stored at the address specified with ColA and RowA. This is done only if link WriteI is one. If link WriteI is zero, the current pixel is skipped. Writing to any address of the valid address range is possible. It is not necessary to write to each memory cell.

2. Read State: The memory is read out sequentially and produces a line of the parameterized width. By use of parameters XOffset, XLength, YOffset and YLength it is possible to define the read address range i.e. a ROI. Reading the memory resets the content of the read addresses to zero. Thus if a memory cell is read which has not been written before, a zero will be output.

The toggling between these two states is triggered by an end-of-frame at the input link I. Please note, that the frame width and height at the output link is fixed to the size defined by parameters XOffset, XLength, YOffset and YLength and is independent from the line width and frame height of the input link.

The operator can be useful for mirroring or sensor correction.

The operator uses the FPGA's internal block RAM memory. Thus, no VisualApplets frame grabber resources of type *RAM* are used. The *FPGA-internal block RAM* is limited. Full resolution frames might not fit into the FPGA-internal block RAM. Consider using operator *FrameBufferRandomRead* instead.

**Operator Restrictions**

- Empty frames are not supported.

  Images with varying line lengths are not supported.

## 27.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image data input<br>WriteI, write enable input<br>ColA, write column address for the pixel at I<br>RowA, write row address for the pixel at I |
| **Output Link** | O, data input |

**Synchronous and Asynchronous Inputs**

- All inputs are synchronous to each other i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

### 27.5.2. Supported Link Format

| Link Parameter | Input Link I | Input Link WriteI | Input Link ColA |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | auto❷ |
| **Arithmetic** | {unsigned, signed} | unsigned | unsigned |
| **Parallelism** | 1 | as I | as I |
| **Kernel Columns** | any | 1 | 1 |
| **Kernel Rows** | any | 1 | 1 |
| **Img Protocol** | VALT_IMAGE2D | as I | as I |
| **Color Format** | any | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | any | FL_NONE | FL_NONE |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

| Link Parameter | Input Link RowA | Output Link O |
|---|---|---|
| **Bit Width** | auto❸ | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | as I | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | as I | parameter FrameWidth |
| **Max. Img Height** | as I | parameter FrameHeight |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ The bit width of the column address is:

$$ColABitWidth = \lceil log_2(FrameHeight) \rceil$$

❸ The bit width of the row address is:

$$RowABitWidth = \lceil log_2(FrameWidth) \rceil$$

### 27.5.3. Parameters

| Implementation | |
|---|---|
| **Type** | static parameter |
| **Default** | SingleBuffer |
| **Range** | {SingleBuffer, DoubleBuffer} |
| This parameter selects the implementation of the FrameMemory (see above). | |

| FrameWidth | |
|---|---|
| **Type** | static parameter |
| **Default** | 1024 |
| **Range** | [0, 65534] |
| This parameter selects the image width of the output link in pixels. | |

| FrameHeight | |
|---|---|
| **Type** | static parameter |
| **Default** | 1024 |
| **Range** | [0, 65534] |
| This parameter selects the image height of the output link in lines. | |

| XOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Height - YLength] |
| This parameter defines the y-coordinate of the upper left corner of the ROI. | |

| XLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [1, Max.Img Width - XOffset] |
| This parameter defines the width of the ROI. | |

| YOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Height - YLength] |
| This parameter defines the y-coordinate of the upper left corner of the ROI. | |

| YLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [1, Max.Img Height - YOffset] |
| This parameter defines the height of the ROI. | |

### 27.5.4. Examples of Use

The use of operator FrameMemory is shown in the following examples:

- Section 13.3, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

## 27.6. Operator FrameMemoryRandomRd

**Operator Library: Memory**

The FrameMemoryRandomRd (frame memory random read) operator buffers an image in memory with random read access. The frame data is transferred into the operator via link I. The random read of the data can be performed by transferring addresses to ports RColA (read column address) and RRowA (read row address). The operator will use the addresses to read the frame data by the given coordinates. The resulting output frame will have the image dimensions of the address inputs.

The required memory size is defined by the image dimension of the input frame (Max. Img Width and Max. Img Height).

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The operator can be implemented in two variants. Namely, a single buffer implementation and a double buffer implementation. The first implementation saves RAM, but does not allow a timely overlap of writing and reading. Thus the address input links RColA and RRowA are stopped during the write state. The latter implementation doubles the RAM and allows to random read the last frame while accepting the next frame at the input link I.

The operator has two states:

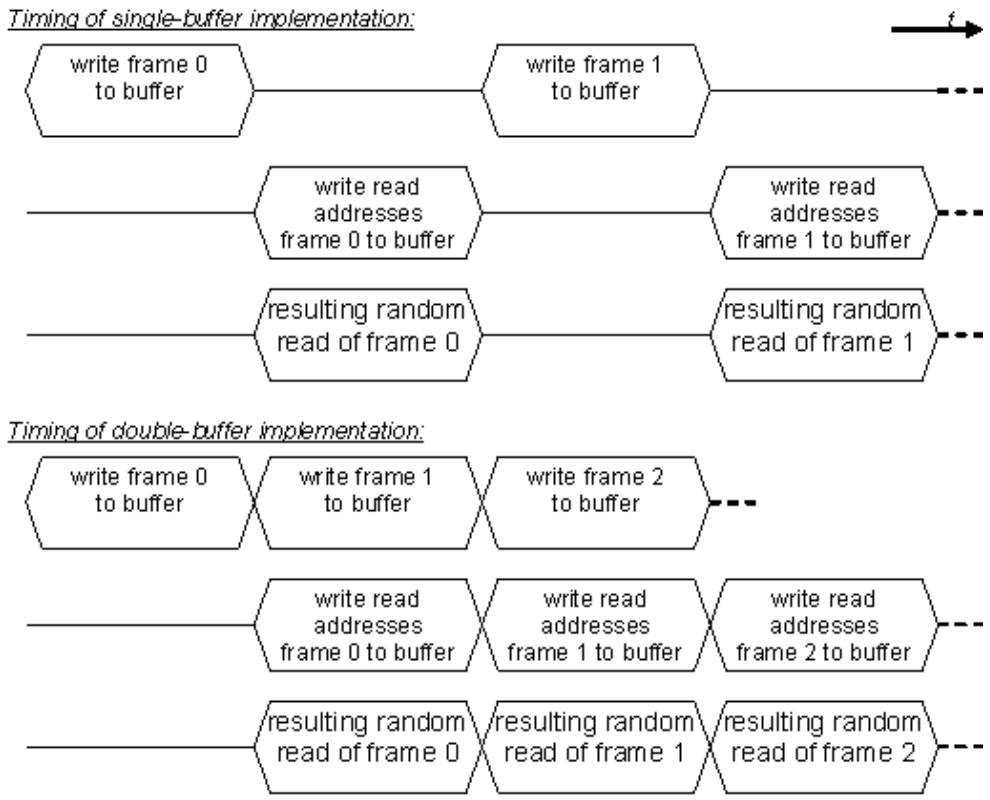
1. Write State: The pixels of link I are stored linear in the buffer i.e. the coordinates of the pixels form the addresses in the buffer.

2. Read State: The memory is read-out using the addresses given at the inputs RColA and RRowA. The number of addresses and address link image dimensions define the image output width and height. For example, if a frame consisting of only one pixel is input at the address inputs, the output frame will only consist of one pixel. The other pixels of the input frame are discarded.

The image data input and the address inputs are not synchronous to each other. They may have different image dimensions. Both address inputs RColA and RRowA have to be O-synchronous.

Please not the timing of the input links. The address inputs must not be sourced by the same operator as the data link input I without buffering. This is because while writing the image data into the operator, no addresses to read the current frame can be accepted. Only if the frame is fully stored into the buffer, addresses can be accepted. In many cases, the operator *CreateBlankImage* is used to generate the images for the addresses.

The timing is visualized in the following figure.




The operator uses the FPGA's internal block RAM memory. Thus, no VisualApplets frame grabber resources of type *RAM* are used. The *FPGA-internal block RAM* is limited. Full resolution frames might not fit into the FPGA-internal block RAM. Consider using operator *FrameBufferRandomRead* instead.

**Operator Restrictions**

- Empty frames are not supported.

  Images with varying line lengths are not supported.

## 27.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image data input<br>RColA, read column address for the pixel at I<br>RRowA, read row address for the pixel at I |
| **Output Link** | O, data input |

**Synchronous and Asynchronous Inputs**

- Synchronous Group: *RColA* and *RRowA*
- Input *I* is asynchronous to the group.

## 27.6.2. Supported Link Format

| Link Parameter | Input Link I | Input Link RColA |
|---|---|---|
| **Bit Width** | [1, 64]❶ | auto❷ |
| **Arithmetic** | {unsigned, signed} | unsigned |

| Link Parameter | Input Link I | Input Link RColA |
|---|---|---|
| **Parallelism** | 1 | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link RRowA | Output Link O |
|---|---|---|
| **Bit Width** | auto❸ | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | as I | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | as RColA | as RColA |
| **Max. Img Height** | as RColA | as RColA |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ The bit width of the column address is:

$$RowABitWidth = \lceil log_2(InputMax.ImageWidth) \rceil$$

❸ The bit width of the row address is:

$$ColABitWidth = \lceil log_2(InputMax.ImageHeight) \rceil$$

## 27.6.3. Parameters

| Implementation | |
|---|---|
| **Type** | static parameter |
| **Default** | SingleBuffer |
| **Range** | {SingleBuffer, DoubleBuffer} |
| This parameter selects the implementation of the FrameMemoryRandomRd (see above). | |

## 27.6.4. Examples of Use

The use of operator FrameMemoryRandomRd is shown in the following examples:

- Section 13.3, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

## 27.7. Operator ImageBuffer

**Operator Library: Memory**

This operator buffers the image stream in the *Frame Grabber RAM (DRAM)*. One VisualApplets resource of type *RAM* is required. Check Section 3.8, 'Allocation of Device Resources' for more information. The operator additionally features region-of-interest (ROI) support. The total number of bits (bit width times parallelism) must not exceed the memory limitations of the respective frame grabber.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The operator works like a FIFO. Any input data is immediately forwarded to the output. However, if the output is blocked, for example, because the following operators cannot process the input bandwidth, the operator will store the data. Thus, the memory will only be filled if the operator cannot output the data. Often, the operator is used to compensate peak input bandwidths.

Internally, the image buffer operates on image lines. This internal line buffer feature results in a very short memory latency. Of course, latency increases if the buffer is filled with more lines. For example, images which are transfered into the memory will be immediately forwarded to the output. In an application, partial camera images can be forwarded to the PC while the camera still transfers the remaining image lines.

Using the parameters *XOffset*, *XLength*, *YOffset* and *YLength* you can define the ROI size. If the input image width is less than the sum of the *XOffset* and *XLength*, the operator will still read the parameterized *XLength*. In this case, the operator will output undefined memory content for the additional pixels. If the input image height is less than the requested output image height, the operator will only output the available lines.

In Line1D application mode, the YOffset and YLength settings do not affect the buffer.

To measure the fill level of the buffer, the operator provides 2 parameters: *FillLevel* and *Overflow*. FillLevel shows the fill level of the RAM in 25% steps. The Overflow parameter is set to 1 when FillLevel is close to or equal to 100% and the next image to be stored in the buffer will exceed the RAM capacity. In case of an overflow, input data is discarded and the input image height is reduced. Thus, incomplete images are stored in the memory. Users have to poll for the overflow parameter. As the duration of the overflow state can be very short, it is possible that it occurs within the polling cycle of the operator.

The parameter *InfiniteSource* is used to specify if the operator is directly connected to a camera or is sequenced with other memory operators. Check Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information.

**Operator Restrictions**

- Empty frames are not supported.



**Available Memory Space**

The operator needs additional memory space for internal data. Thus, not the full amount of the used RAM resource can be used for buffering image data. The space actually available for buffered image data depends on the hardware platform (*Platform RAM Size* and *Platform RAM Interface Width*) and on the link configuration. For information on *Platform RAM Size* and *Platform RAM Interface Width*, see Appendix A, '*Device Resources*'.)

You can calculate the available capacity of the RAM resource using the following formulas:

- Utilized RAM Size [in Bytes] = ((PRS * 2^20 / (PRIW / 8 bit)) / RLS) * MIW * BW / 8
- Utilized RAM Size [in Lines] = ((PRS * 2^20 / (PRIW / 8 bit)) / RLS)
- Utilized RAM Size [in Frames] = ((PRS * 2^20 / (PRIW / 8 bit)) / RLS) / MIH

Abbrevations used in the formulas and units of measure:

- **PRS**: Platform RAM Size [in MB, 1MB = 2^20 Byte]

- **PRIW**: Platform RAM Interface Width [in bit]
- **MIW**: MaxImageWidth [in pixel]
- **MIH**: MaxIMageHeight [in lines]
- **P**: Parallelism
- **BW**: BitWidth [in bit]
- **RLS**: Raw Line Size: 2 ^ ceil(log2n(MIW / P + 4))

### 27.7.1. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width.



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

### 27.7.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

### 27.7.3. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶❷ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any❸ | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | 65535 | as I |
| **Max. Img Height** | 65535 | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷❸ The product of the bit width and the parallelism must not exceed the native ram data width. Check Appendix A, '*Device Resources*' for more information.

### 27.7.4. Parameters

| **XOffset** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Width - XLength] |
| This parameter defines the x-coordinate of the upper left corner of the ROI.<br>The step size is the parallelism. | |

| **XLength** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [2*parallelism, Max.Img Width - XOffset] |
| This parameter defines the width of the ROI.<br>The step size is the parallelism. | |

| **YOffset** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Height - YLength] |
| This parameter defines the y-coordinate of the upper left corner of the ROI. | |

| **YLength** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [1, Max.Img Height - YOffset] |
| This parameter defines the height of the ROI. | |

| **FillLevel** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of DRAM in 25% steps. | |

| **Overflow** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| This parameter indicates a buffer overflow. | |

| **InfiniteSource** | |
|---|---|
| **Type** | static parameter |
| **Default** | ENABLED |
| **Range** | {ENABLED, DISABLED} |

| **InfiniteSource** |
| --- |
| This parameter activates support for infinite source operators like Camera operators. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. |

## 27.7.5. Examples of Use

The use of operator ImageBuffer is shown in the following examples:

- 2. *Getting Started*

  Getting Started
- Figure 3.1, 'Simple VisualApplets Design'

  Basic Principles - Learn the Idea of VisualApplets
- Section 3.5, 'Data Flow '

  Data Flow - Learn about the Pipeline Structure used in VisualApplets
- Section 3.6.2, 'O-Type Networks'

  Synchronization Rules - The use of the operator in an O-type Network.
- Section 3.6.9, 'Infinite Sources / Connecting Cameras'

  Infinite Sources - Connecting operators to cameras. (DRC2 Latency Error)
- Section 3.7.1, 'Module Properties'

  Design Parameterization
- Disabled Parameters

  Design Parametrization - Disabled Parameters
- Section 3.7.1.5, 'Illegal Parameter Value States'

  Design Parametrization - Illegal Parameter Value
- Section 3.8, 'Allocation of Device Resources'

  Learn the allocation of the device resources of the operator.
- Figure 9.4, 'Illegal Condition after Link Property Change'

  Tutorial Basic Acquisition - Illegal Condition at *ImageBuffer operator*
- Section 12.1.2, 'JPEG Encoder Gray'

  Examples - A simple example which shows the usage of the JPEG operators.
- Section 12.1.4, 'JPEG Compression Using Operator **JPEG_Encoder**'

  Examples - Simple examples which show the usage of the operator **JPEG_Encoder**.
- Section 12.1.5, 'JPEG Color Compression Using User Library Elements'

  Examples - Simple examples which shows the usage of the JPEG user library elements for color JPEG compression.
- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.3.3, 'Blob2D ROI Selection'

  Examples - The blob analysis operator is applied to an input camera image. The applet shows the usage of the blob data in the applet. In this case, the object with the maximum are is localized and the coordinates are used to cut out the object from the original image.

- Section 12.6.2, 'Image Dimension Test'

  Example - The image dimension is measured and can be used to analyze the design flow.

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

- Section 12.6.4, 'Manual Image Injection'

  Example - For debugging purposes images can be inserted manually.

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

- Section 12.8.4.4, 'Filter for Line Scan Cameras'

  Examples - Explains how to implement a filter for line scan cameras.

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

## 27.8. Operator ImageBufferMultiRoI

**Operator Library: Memory**

This operator provides support for multiple regions of interest (ROI) for each buffered image. All ROIs are read out sequentially. Each ROI is a new and independent output image: ROI 0, ROI 1, ROI 2 and so on until ROI N-1. N is the number of ROIs specified by the parameter *NumRoI*. Supported are all possible rectangular regions as long as the frame dimensions are not exceeded, i.e., a single pixel, a single line, a single column, a rectangular region, or the complete frame can be defined as a ROI. Different ROIs can be of different size. Each ROI can be defined individually. ROIs can overlap each other. However, all ROIs must meet the restriction of the maximal image size defined by the input and output links, i.e. ROI widths must not exceed the maximal image width and ROI heights must not exceed the maximal image height.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The operator buffers the image stream in the frame grabber on board RAM. One VisualApplets resource of type *RAM* is required (see Section 3.8, 'Allocation of Device Resources').

ImageBufferMultiRoI operates on images of fixed size defined by the input link maximal image width and height parameters. This implies input frame dimension control performed by the operator.

Images that are smaller than the maximal image dimension will be buffered without modifications. However the reading of ROIs could cause the read of undefined pixel values i.e. pixel values outside of the input image dimension. The missing frame pixels contain random data and should be treated in further processing as undefined.

In case of an overflow, i.e. the buffer cannot store any more images due to being full, all incoming images will be dropped until there is enough free space in the buffer to store at least one frame. An overflow can occur because the reading of the ROIs is to slow in comparison to the writing. This can either be because to many ROIs are defined or because successive operators block the output. In normal situations of balanced designs an overflow will never occur.

The operator provides ROI images on its output as soon as the correspondent frame is completely buffered. The type of reading out is defined by the parameter *Mode*.

The parameter *MaxNumRoI* specifies the maximal number of ROIs. The actual number of ROIs used can be less and is defined using parameter *NumRoI*.

The parameter *NumRoI* specifies the number of ROIs the operator will provide at the output for each buffered image. This number mus not exceed *MaxNumRoI* when set to the dynamic type.

The parameter *Mode* specifies the operator behavior for processing images. **`FreeRun`** defines the buffer to provide ROIs at its output as soon as possible, i.e. as soon as the correspondent frame is completely buffered. This mode is also called free running mode. The operator is controlled only by the pipeline flow control mechanisms. **`WaitAfterImage`** mode forces the operator to hold on after all ROIs of a frame were provided on its output. To restart reading of the ROIs of the next frame, the parameter *Unlock* has to be written. **`WaitAfterRoI`** is similar to **`WaitAfterImage`** except that the operator stops after each ROI of each frame. To restart further processing a write cycle to the parameter *Unlock* is required.

The parameter *Unlock* is used for the modes **`WaitAfterImage`** and **`WaitAfterRoI`** to unlock the operator and to start further read out operations. In mode **`FreeRun`** this parameter has no effect.

XOffset, XLength, YOffset and YLength specify a set of ROIs. These parameters are field parameters. The parameter type of these parameters can be set to dynamic or static, this means adjustable or not adjustable during runtime. The parameter type of all ROI parameters is automatically adapted accordingly, when the parameter type of one ROI parameter is set. The size of the field is defined by parameter *MaxNumRoI*. The operator will read *NumRoI* ROIs in sequential order. Learn on how to configure field parameters in Section 3.7.1.3, 'Parameter Editing'.

All ROIs can be updated dynamically. However, the updating can be performed for the modes FreeRun, WaitAfterImage and WaitAfterRoI only if no image acquisition is started. Additionally in the modes

WaitAfterImage and WaitAfterRoI, ROIs can be updated if the corresponded frame / all ROIs of the frame were transmitted and the operator is waiting for a write cycle to the Unlock parameter.

Updating ROIs during image acquisition can lead to temporarily non-rectangular ROIs and should be avoided.

If the operator is used with image protocol VALT_LINE1D, the Y coordinate parameters are disabled. In 1D operation, the operator will read one ROI as one line. The output of the operator will then be an infinite 1D image stream, where each line is represented by one ROI e.g. line 0 = ROI 0, line 1 = ROI 1, line N = ROI N, line N+1 = ROI 0, ...

To measure the fill level of the buffer the operator provides 2 registers: *FillLevel* and *Overflow*. *FillLevel* shows the percental fill level of the RAM. The *Overflow* parameter is set to 1 when *FillLevel* is close or is 100% and the next image to be stored in the buffer will exceed the RAM capacity.

Parameter *InfiniteSource* is used to specify if the operator is directly connected to a camera or is sequenced with other memory operators. Check Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information.

**Operator Restrictions**

- Empty frames are not supported.
- Images with varying line lengths are not supported.



***ImageBufferMultiRoI* Is Not Available for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta Platforms**

*ImageBufferMultiRoI* is not supported on imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms, but you can use an equivalent user library element for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms instead. See Section 4.2.8, 'Delivered User Libraries' for instructions how to work with user library elements.



**Example: Optimization of Memory Usage**

A cumstomer feeds an image with the dimensions 10000 x 7096 pixels (link property on input port). The parallelism is set to 4. The pixel width is set to 8bpp.

The platform used in this example is mE4, i.e. the platform provides 128 Mbyte RAM banks. To use all 128 MB, the operator should be fed with 64 bit. But with parallelism 4, only 32 bits ( 4 x 8 bits) are fed, i.e., the user can only use 64 MB of the RAM. To save an image with the dimensions of 10000 x 7096 pixels, ca. 70.960.000 Bytes (ca. 70 MB) are required, which is more than the available 64 MB. This is detected by VisualApplets, and the link is highlighted in red.

Solution: Parup (to 8) in front of the operator ImageBufferMultiRoi.

## 27.8.1. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width.



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

### 27.8.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

### 27.8.3. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶❷ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any❷ | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any❸ | as I |
| **Max. Img Height** | any❸ | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ The product of the bit width and the parallelism must not exceed the native ram data width (*RamDataWidth*). Check Appendix A, '*Device Resources*' for more information.

❸ Maximum image dimensions for I:

- *I.Img.Protocol* = VALT_IMAGE2D: 2^*RamAddressWidth* >= (*I.Max.Img.Width* * *I.Max.Img.Height*) / *I.Parallelism*
- *I.Img.Protocol* = VALT_LINE1D: 2^*RamAddressWidth* >= *I.Max.Img.Width* / *I.Parallelism*

### 27.8.4. Parameters

| **RamDataWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of data bits that can be used at the RAM interface. It's the maximum number of bits for input and output. | |

| **RamAddressWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of address bits available. This helps to calculate the maximum allowed image dimensions. | |

| **MaxNumRoI** | |
|---|---|
| **Type** | static parameter |

| MaxNumRoI | |
|---|---|
| **Default** | 1 |
| **Range** | [1, 65535] |
| This parameter defines the maximum number of ROIs which the operator is capable to store. | |

| NumRoI | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | [1, MaxNumRoI] |
| This parameter defines the number of ROIs actually used. The operator will read NumRoI ROIs from each input image and provide them on its output. If the parameter is set to static the range is [1, 65536] and MaxNumRoI is disabled. | |

| Mode | |
|---|---|
| **Type** | static parameter |
| **Default** | FreeRun |
| **Range** | {FreeRun, WaitAfterImage, WaitAfterRoI} |
| This parameter defines the operation mode of the read out algorithm. In FreeRun mode the operator provides ROI images as soon as the correspondent frame is completely buffered. In WaitAfterImage mode the operator will provide all ROIs within a single frame, then stop and wait for a software access to write on Unlock parameter to enable reading out of the next frame ROIs. In WaitAfterRoI mode the operator will stop after each ROI of each frame. To restart the reading out operation, the software has to write to the Unlock parameter. | |

| Unlock | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1 |
| **Range** | {1} |
| This parameter implements the handshake for the mode WaitAfterImage and WaitAfterRoI. Writing to this parameter value 1 will cause the operator to continue reading out of the next ROI / all ROIs of the next Frame.<br><br>This parameter can only be used of parameter Mode is set to WaitAfterImage or WaitAfterRoI. | |

| XOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Width - XLength] |
| This field parameter defines the x-coordinate of the upper left corner of the ROI.<br><br>The step size is the parallelism.<br><br>Learn on how to configure field parameters in Section 3.7.1.3, 'Parameter Editing'. | |

| XLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [2*parallelism, Max.Img Width - XOffset] |
| This field parameter defines the width of the ROI.<br><br>The step size is the parallelism.<br><br>Learn on how to configure field parameters in Section 3.7.1.3, 'Parameter Editing'. | |

| YOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Height - YLength] |
| This field parameter defines the y-coordinate of the upper left corner of the ROI.<br><br>Learn on how to configure field parameters in Section 3.7.1.3, 'Parameter Editing'. | |

| YLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [1, Max.Img Height - YOffset] |
| This field parameter defines the height of the ROI.<br><br>Learn on how to configure field parameters in Section 3.7.1.3, 'Parameter Editing'. | |

| FillLevel | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of DRAM. | |

| Overflow | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| This parameter indicates a buffer overflow. | |

| InfiniteSource | |
|---|---|
| **Type** | static parameter |
| **Default** | ENABLED |
| **Range** | {ENABLED, DISABLED} |
| This parameter activates support for infinite source operators like Camera operators. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

## 27.8.5. Examples of Use

The use of operator ImageBufferMultiRoI is shown in the following examples:

- Section 12.4.2.3, 'Color Plane Separation Option 3 - Sequential with Operator ImageBufferMultiRoI'

  Sequential DMA output of the color planes. The color separations is performed using operator ImageBufferMultiROI.

- Section 12.4.2.4, 'Color Plane Separation Option 4 - Sequential with Operator ImageBufferMultiRoI and a pre-sort of the Color Planes'

  Sequential DMA output of the color planes. The color separations is performed using operator ImageBufferMultiROI. An additional pre-sorting optimizes the bandwdith and resources.

## 27.9. Operator ImageBufferMultiRoIDyn

**Operator Library: Memory**

This operator provides support for multiple dynamic regions of interest (ROI) for each buffered image. The ROI coordinates are defined using additional input links.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

All ROIs are read sequentially as individual images: ROI 0, ROI 1, ROI 2 and so on until ROI N-1. N is the maximal number of allowed ROIs. Supported are all possible rectangular regions as long as the frame dimensions are not exceeded, i.e., a single pixel, a single line, a single column, a rectangular region, or the complete frame can be defined as a ROI. Different ROIs do not need to have the same size. Each ROI can be defined individually. ROIs can overlap each other. ROIs violating the maximal image dimensions are filtered out by the operator, i.e., they will be skipped. Also, all illegal ROIs are skipped, e.g., ROIs with negative height or width.

The operator buffers the image stream in the *Frame Grabber RAM (DRAM)*. One VisualApplets resource of type *RAM* is required (see Section 3.8, 'Allocation of Device Resources').

ROI coordinates are provided on separate operator input links as images containing the ROI coordinates. Each pixel in these images is treated by the operator as a valid ROI coordinate set. Thus, the size of the ROI input images define the number of ROIs. For each input image at input link I, a new ROI coordinate image set is required. The ROI input consists of 4 coordinate input links. Respectively, links for X and Y coordinates of the top left ROI corner as well as links for X and Y coordinates of the bottom right ROI corner.

Note that ROI X-coordinates are transformed to meet the x-granularity of the input link I defined by its parallelism, i.e., the operator ImageBufferMultiRoIDyn can only cut lines with the granularity of the parallelism. The coordinate XTopLeft has to be the index of the first component in the parallel word. The coordinate XBottomRight has to be the index of a last pixel in a parallel word.

Let's have a look at an example to explain the operator behavior: Suppose a parallelism of four at the input link I. Therefore XTopLeft has to be: 0, 4, 8, 12, ... XBottomRight has to be 3, 7, 11, 15, ... If the coordinates do not match with these constraints, they are rounded down (XTopLeft) or up (XBottomRight). For example, with the ROI x-coordinates 3 and 7, the first column of the ROI is column number 3 and the last column is column number 7 of the input image. The operator rounds down XTopLeft to 0; XBottomRight remains 7; Thus, the width becomes 8. The ROI height is not dependent on the parallelism and thus can be of any legal value.



**Formula:**

XTopLeft* = floor(XTopLeft / Parallelism) * Parallelism

XBottomRight* = ceil((XBottomRight + 1) / Parallelism) * Parallelism - 1



**Examples:**

With a parallelism of four at the input link I:

- 112 to 551 becomes 112 to 551. Therefore the width becomes = 551 + 1 - 112 = 440
- 101 to 540 becomes 100 to 543. Therefore the width becomes = 543 + 1 - 100 = 444

All 4 ROI inputs are synchronous to each other, i.e., the images on these links must be of the same size and provided at the same time synchronously. This is always the case when they are sourced by the same M type module through an arbitrary network of O-type module. See the synchronization rules for more information (Section 3.6.4, 'M-type Operators with Multiple Inputs'). The ROI coordinates are asynchronous to the input link I.

The operator starts reading the ROIs as soon as the image and all ROI coordinates are fully transfered into the buffer. In practice, the ROI coordinates can be calculated and determined from the input

image but there is no necessity. Users can also use for example operator CreateBlankImage for ROI coordinate generation.

When the ROI image is empty, i.e. does not contain any pixels, the operator will provide an empty image on its output. An empty image contains no pixels. If any ROI is illegal the operator will provide an empty image for that ROI.

ImageBufferMultiRoIDyn operates on images of fixed size defined by the input link maximal image width and height parameters. This implies input frame dimension control performed by the operator.

Images that are smaller than the maximal image dimension will be buffered without modifications. However, the reading out of ROIs will be performed on a frame of maximal possible image dimension. Missing frame pixels contain random data and should be treated in further processing as undefined.

Images that exceed either the maximal frame height or the maximal frame width will be cut to the maximal frame dimension and buffered as such. Reading out of ROIs will be based again on the frame of maximal dimension.

When an overflow occurs, i.e. the buffer cannot store more images due to being full, all incoming images will be dropped until there is enough free space in the buffer to store at least one frame. Overflow can occur due to different reasons but all of them result in the reading out of being performed too slow in comparison to writing. This can either be because of defining too many ROIs or because of a blocking condition caused by the further processing pipeline or because of a too fast input image stream. In normal situations of balanced designs an overflow will never occur.

To measure the fill level of the buffer the operator provides 2 registers: FillLevel and Overflow. FillLevel shows the percentage fill level of the RAM. The Overflow parameter is set to 1 when FillLevel is close or is 100% and the next image to be stored in the buffer will exceed the RAM capacity.

The operator provides ROI images on its output as soon as the correspondent frame is completely buffered and all ROI sets for the current image were received on the ROI inputs.

The maximal amount of supported ROIs is determined by the product of the maximal image width and height settings of the XTopLeft input link. The maximal allowed number of ROIs is 2E20-1. The buffering of the ROI coordinates requires *FPGA-internal block RAM*. Therefore, the number of ROIs is limited by the available FPGA-internal block RAM.

Parameter *InfiniteSource* is used to specify if the operator is directly connected to a camera or is sequenced with other memory operators. Check Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information.

**Operator Restrictions**

- Empty frames or empty lines on input link I are not supported.
- Images with varying line lengths on port I are fully supported.
- Empty frames or empty lines on ROI ports are fully supported.
- Images with varying line lengths on ROI ports are fully supported.



***ImageBufferMultiRoiDyn* Is Not Available for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta Platforms**

*ImageBufferMultiRoiDyn* is not supported on imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms, but you can use an equivalent user library element for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms instead. See Section 4.2.8, 'Delivered User Libraries' for instructions how to work with user library elements.

## 27.9.1. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width.



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

## 27.9.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image data input<br>XTopLeft, coordinate data input<br>YTopLeft, coordinate data input<br>XBottomRight, coordinate data input<br>YBottomRight, coordinate data input |
| **Output Link** | O, data input |

**Synchronous and Asynchronous Inputs**

- The 4 ROI inputs are synchronous to each other.
- The ROI inputs are asynchronous to input *I*.

## 27.9.3. Supported Link Format

| Link Parameter | Input Link I | Input Link XTopLeft | Input Link YTopLeft |
|---|---|---|---|
| **Bit Width** | [1, 64]❶❷ | auto❸ | auto❹ |
| **Arithmetic** | {unsigned, signed} | unsigned | unsigned |
| **Parallelism** | 2^N, with N ={0,1,2...}❷ | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | VALT_IMAGE2D |
| **Color Format** | any | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | any | FL_NONE | FL_NONE |
| **Max. Img Width** | any❺ | 2E20-1❻ | as XTopLeft |
| **Max. Img Height** | any❺ | 2E20-1❻ | as XTopLeft |

| Link Parameter | Input Link XBottomRight | Input Link YBottomRight | Output Link O |
|---|---|---|---|
| **Bit Width** | auto❸ | auto❹ | as I |
| **Arithmetic** | unsigned | unsigned | as I |
| **Parallelism** | 1 | 1 | as I |
| **Kernel Columns** | 1 | 1 | as I |
| **Kernel Rows** | 1 | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | as I |

| Link Parameter | Input Link XBottomRight | Input Link YBottomRight | Output Link O |
|---|---|---|---|
| **Color Format** | VAF_GRAY | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | FL_NONE | as I |
| **Max. Img Width** | as XTopLeft | as XTopLeft | as I |
| **Max. Img Height** | as XTopLeft | as XTopLeft | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The product of the bit width and the parallelism must not exceed the native RAM data width (*RamDataWidth*). Check Appendix A, '*Device Resources*' for more information.
❸ The bit width is

$$BitWidth = CEIL(log_2(MaxImageWidth(I) - 1))$$

❹ The bit width is

$$BitWidth = CEIL(log_2(MaxImageHeight(I) - 1))$$

❺ The maximum image dimension must not exceed the size of the available RAM: 2^*RamAddressWidth* >= (*I.Max.Img.Width* * *I.Max.Img.Height*) / *I.Parallelism*
❻ The maximum number of pixels for ROI images (consequenlty the maximum number of ROIs per image) must not exceed 2^20-1: 2^20 > *XTopLeft.Max.Img.Width* * *XTopLeft.Max.Img.Height*

## 27.9.4. Parameters

| **RamDataWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of data bits that can be used at the RAM interface. It's the maximum number of bits for input and output. | |

| **RamAddressWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of address bits available. This helps to calculate the maximum allowed image dimensions. | |

| **FillLevel** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of DRAM in 25% steps. | |

| **Overflow** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| This parameter indicates a buffer overflow. | |

| **InfiniteSource** | |
|---|---|
| **Type** | static parameter |

| InfiniteSource | |
|---|---|
| **Default** | ENABLED |
| **Range** | {ENABLED, DISABLED} |
| This parameter activates support for infinite source operators like Camera operators. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

### 27.9.5. Examples of Use

The use of operator ImageBufferMultiRoIDyn is shown in the following examples:

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

# 27.10. Operator ImageBufferSC

**Operator Library: Memory**

This operator features tap sorting (=sensor correction), which is often required if line scan cameras are used. Moreover, a region-of-interest (ROI) support is included. The total number of bits (bit width times parallelism) must not exceed the memory limitations of the respective frame grabber.

This operator buffers the image stream the in *Frame Grabber RAM (DRAM)*. One VisualApplets resource of type *RAM* is required (see Section 3.8, 'Allocation of Device Resources').

The tap sorting feature enables transparent support for sensors, whose readout scheme does not follow a sequential raster-scan format. Supported are the most popular readout schemes for dual tap base cameras (when using a parallelism of 4) and quad tap medium cameras.

For other parallelism, the operator can be used in combination with other operators.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The operator works like a FIFO. Any input data is immediately forward to the output. However, if the output is blocked for example because the following operators cannot process the input bandwidth, the operator will store the data. Thus the memory will only be filled, if the operator cannot output the data. Often, the operator is used to compensate peak input bandwidths.

Internally, the image buffer operates on image lines. This internal line buffer feature results in a very short memory latency. Of course, latency increases if the buffer is filled with more lines. For example, images which are transfered into the memory will be immediately forwarded to the output. In an application, partial camera images can be forwarded to the PC while the camera still transfers the remaining image lines.

Using parameters *XOffset*, *XLength*, *YOffset* and *YLength* the ROI size can be defined. If the input image width is less than the sum of the *XOffset* and *XLength*, the operator will still read the parameterized *XLength*. In this case, the operator will output undefined memory contend to the exceeding pixel. If the input image height is less than the requested output image height, the operator will only output the available lines.

In Line1D application mode, the YOffset and YLength settings do not affect the buffer.

To measure the fill level of the buffer the operator provides 2 parameters: *FillLevel* and *Overflow*. FillLevel shows the percentaged fill level of the RAM in 25% steps. The Overflow parameter is set to 1 when FillLevel is close to or is 100% and the next image to be stored in the buffer will exceed the RAM capacity. In case of an overflow, input data is discarded and the input image height is reduced. Thus, incomplete images are stored in memory. Users have to poll for the overflow parameter. As the duration of the overflow state can be very short it is possible that it is in between of a polling cycle of the operator.

Parameter *InfiniteSource* is used to specify if the operator is directly connected to a camera or is sequenced with other memory operators. Check Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information.

**Operator Restrictions**

- Empty frames are not supported.

  Images with varying line lengths are not supported.

  The input line width has to be divisible by the number of taps multiplied with the parallelism.

## 27.10.1. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width.



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

## 27.10.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

## 27.10.3. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] unsigned, [2, 64] signed❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | 4❷ | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | 65535 | as I |
| **Max. Img Height** | any | as I |

❶❷ The product of the bit width and the parallelism must not exceed the native RAM data width. Check Appendix A, '*Device Resources*' for more information.

## 27.10.4. Parameters

| XOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Width - XLength] |
| This parameter defines the x-coordinate of the upper left corner of the ROI.<br><br>The step size is the parallelism. | |

| XLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [2*parallelism, Max.Img Width - XOffset] |

| XLength |
|---|
| This parameter defines the width of the ROI.<br><br>The step size is the parallelism. |

| YOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Height - YLength] |
| This parameter defines the y-coordinate of the upper left corner of the ROI. | |

| YLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [1, Max.Img Height - YOffset] |
| This parameter defines the height of the ROI. | |

| SensorCorrection | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | SMODE_UNCHANGED |
| **Range** | |

Available options are

| | |
|---|---|
| SMODE_UNCHANGED | \|-> \| |
| SMODE_REVERSE | \| <-\| |
| SMODE_TAP2_0 | \|1> 2> \| |
| SMODE_TAP2_1 | \| <2 <1\| |
| SMODE_TAP2_2 | \|1> <2\| |
| SMODE_TAP4_0 | \|1> 2> 3> 4> \| |
| SMODE_TAP4_1 | \| <4 <3 <2 <1\| |
| SMODE_TAP4_2 | \|12> <34\| |
| SMODE_TAP4_3 | \| <1 <2 <3 <4\| |
| SMODE_TAP4_4 | \|12> <43\| |
| SMODE_TAP4_5 | \|1> 2> <3 <4\| |
| SMODE_TAP4_6 | \|12> 34> \| |

The parameter specifies the sensor readout scheme of the connected camera. SMODE_UNCHANGED and SMODE_REVERSE support single, dual and quad tap cameras. SMODE_TAPS2_x support dual tap (base) cameras. SMODE_TAP4_x support quad tap (medium) cameras.

| FillLevel | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of DRAM in 25% steps. | |

| Overflow | |
|---|---|
| **Type** | dynamic read parameter |

| Overflow | |
|---|---|
| **Default** | 0 |
| **Range** | [0, 1] |
| This parameter indicates a buffer overflow. | |

| InfiniteSource | |
|---|---|
| **Type** | static parameter |
| **Default** | ENABLED |
| **Range** | {ENABLED, DISABLED} |
| This parameter activates support for infinite source operators like Camera operators. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

### 27.10.5. Examples of Use

The use of operator ImageBufferSC is shown in the following examples:

- Section 13.3, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

# 27.11. Operator ImageBufferSpatial

**Operator Library: Memory**

| **Availability for Hardware Platforms** |
| --- |
| Please note that this operator is only available for target platforms of the microEnable 4 series (including PixelPlant). |

This operator computes a spatial correction of its color components. Moreover, a region-of-interest (ROI) feature is included. The total number of bits (bit width x parallelism) must not exceed the memory limitations of the respective frame grabber.

This operator buffers the image stream in the *Frame Grabber RAM (DRAM)*. One VisualApplets resource of type *RAM* is required (see Section 3.8, 'Allocation of Device Resources').

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The operator's key feature is the correction of the spatial shift of tri-linear line scan color cameras. For each color component red, green, and blue a relative shift can be defined using the parameters DeltaRed, DeltaGreen, and DeltaBlue. A shift means the delay in lines between the input and the resulting output image. A shift of 0 leaves the input image untouched. If for example DeltaRed = 0, DeltaGreen = 1, and DeltaBlue = 2, the first line of the output image consists in the following components: Red component read from the first input image line, green component is read from the second input image line, and blue component read from the third input image line. In order to switch the readout direction from forward to backward, just change the parameters: DeltaRed = 2, DeltaGreen = 1, and DeltaBlue = 0. Delta shifts can be negative, 0 or positive.

The buffer transforms delta values to relative deltas internally. The transformation implements this formula:

$$delta^* = delta - Max\left(delta_{red}, delta_{green}, delta_{blue}\right)$$

where delta is DeltaRed, DeltaGreen or DeltaBlue. This transformation is performed internally. The delta parameters of the operator stay unchanged and show the specified values. The transformation to relative deltas causes one delta offset to become 0 and the other two to become either 0 or negative, e.g. the configuration [DeltaRed = 1, DeltaGreen = 2, DeltaBlue = 1] is equivalent to [DeltaRed = -1, DeltaGreen = 0, DeltaBlue = -1]. The configuration [DeltaRed = -3, DeltaGreen = -3, DeltaBlue = -3] and [DeltaRed = 1, DeltaGreen = 1, DeltaBlue = 1] are both equivalent to [DeltaRed = 0, DeltaGreen = 0, DeltaBlue = 0]. A configuration like [DeltaRed = 1, DeltaGreen = 1, DeltaBlue = 1] indicates to perform a spatial correction with the configuration [DeltaRed = 0, DeltaGreen = 0, DeltaBlue = 0] and cut off the first line. Since the operator supports only relative delta shifts, the cut off of the first line will not be performed by the operator. However it is relatively easy to achieve the cutting outside the operator by using VA operators like CoordinateY and RemoveLine.

For each read out line the operator provides a line ID marker that is synchronous to the output link O. The line ID is a global marker and can be used to detect when lines are lost due to an overflow in the buffer. When an overflow occurs the buffer will discard input lines until the overflow condition is lifted, i.e., the RAM has enough storage room for at least 1 RoI line. The difference between 2 consecutive lines minus 1 is the amount of lines lost due to an overflow in the buffer.

The operator will continuously correct spatial shifts - a typical requirement for many line scan applications. However when the acquisition starts the buffer needs at least a certain amount of lines to output a corrected line. When this threshold is not reached no lines are output. The buffer outputs only lines that were corrected. The start-up lines required for the 1st image line to be corrected are not output. The threshold value depends on the specified delta shifts, e.g. DeltaRed = 0, DeltaGreen = 1, DeltaBlue = -3 require 5 lines to perform a correction. Thus the threshold is 5. After the 5th line, the buffer starts to output lines. In other words, the buffer starts to output lines as soon as it can correct them.

In frame correction applications (VALT_IMAGE2D input) the spatial correction will be performed on each frame individually, i.e. for every frame the spatial correction begins anew.

The first line output by the buffer owns always the line ID 0. In 2D application mode the line ID of the 1st frame line will show a gap to the last line of the previous frame. The gap size equals the amount

of lines required to perform correction for this 1st output line. Not correctable lines are suppressed by the buffer.

Using parameters *XOffset* and *XLength* the ROI size can be defined. If the XLength is not divisible by the link parallelism the operator will insert dummy pixels to fill up the last parallel word, e.g. the link parallelism is 2, XOffset is 0 and XLength is 7, the operator will output 4 parallel words each consisting of 2 pixels. The last word will contain a dummy pixel. The value of that dummy pixel is undefined. In VA simulation dummy pixels will be set to zero for better visibility. The sum of XOffset and XLength parameters must not exceed the maximal link image width. If the input image width is less than the sum of the x-offset and the x-length, the operator will still read the parameterized x-length. In this case, the operator will output undefined memory contend to the exceeding pixel.

To measure the fill level of the buffer the operator provides 2 parameters: *FillLevel* and *Overflow*. FillLevel shows the percentaged fill level of the RAM in 25% steps. The Overflow parameter is set to 1 when FillLevel is close to or is 100% and the next image to be stored in the buffer will exceed the RAM capacity. In case of an overflow, input data is discarded and the input image height is reduced. Thus, incomplete images are stored in memory. Users have to poll for the overflow parameter. As the duration of the overflow state can be very short it is possible that it is in between of a polling cycle of the operator.

Parameter *InfiniteSource* is used to specify if the operator is directly connected to a camera or is sequenced with other memory operators. Check Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information.

**Operator Restrictions**

- Empty frames are not supported.

  Images with varying line lengths are not supported.

## 27.11.1. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width.



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

## 27.11.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Links** | O, data input<br>LinkID, data input |

## 27.11.3. Supported Link Format

| Link Parameter | Input Link I | Output Link O | Output Link LinkID |
|---|---|---|---|
| **Bit Width** | [3, 63] unsigned, [6, 63] signed❶ | as I | auto |

| Link Parameter | Input Link I | Output Link O | Output Link LinkID |
|---|---|---|---|
| **Arithmetic** | {unsigned, signed} | as I | unsigned |
| **Parallelism** | any❷ | as I | as I |
| **Kernel Columns** | 1 | as I | as I |
| **Kernel Rows** | 1 | as I | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I | as I |
| **Color Format** | VAF_COLOR | as I | VAF_GRAY |
| **Color Flavor** | FL_RGB | as I | FL_NONE |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶❷ The product of the bit width and the parallelism must not exceed the native ram data width. Check Appendix A, '*Device Resources*' for more information.

## 27.11.4. Parameters

| **XOffset** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Width - XLength] |
| This parameter defines the x-coordinate of the upper left corner of the ROI. | |

| **XLength** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [1, Max.Img Width - XOffset] |
| This parameter defines the width of the ROI. | |

| **DeltaRed** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | any |
| This parameter defines the shift of the red component. The delta can be a positive or negative integer value. | |

| **DeltaGreen** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | any |
| This parameter defines the shift of the green component. The delta can be a positive or negative integer value. | |

| **DeltaBlue** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | any |
| This parameter defines the shift of the blue component. The delta can be a positive or negative integer value. | |

| FillLevel | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of DRAM in 25% steps. | |

| Overflow | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| This parameter indicates a buffer overflow. | |

| InfiniteSource | |
|---|---|
| **Type** | static parameter |
| **Default** | ENABLED |
| **Range** | {ENABLED, DISABLED} |
| This parameter activates support for infinite source operators like Camera operators. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

# 27.12. Operator ImageFifo

**Operator Library: Memory**

This operator can be used to buffer a relatively small number of pixels in an FPGA based memory. The operator does not require a VisualApplets resource of type *RAM*. Instead, the operator uses FPGA internal memory. This can either be the *FPGA-internal block RAM*, the *FPGA distributed RAM (LUT RAM)*, or *UltraRAM memory* (URAM is only available for the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms). Which of the two is used can be set manually via Parameter *ImplementationType* or is selected automatically by the operator depending on the operator's configuration.

The ImageFifo operator is often used to buffer lines or pixels before a synchronization. As the FPGA internal memory is limited, it is unlikely that frames of full resolution can be buffered without exceeding the available resources.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

Parameters *EntitiesToStore* and *EntityType* define the maximum available buffer size. The operator is a first input, first output memory (FIFO). Any input data is immediately forward to the output. However, if the output is blocked for example because the following operators cannot process the input bandwidth, the operator will store the data. Thus the memory will only be filled, if the operator cannot output the data.

To measure the fill level of the buffer the operator provides the parameter *FillLevel*. FillLevel shows the percentage fill level of the Fifo.

Parameter *InfiniteSource* is used to specify if the operator is directly connected to a camera or is sequenced with other memory operators. If InfiniteSource is disabled, the operator cannot get into an overflow condition. Check Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information.

Parameter *ImplementationType* allows to specify the hardware memory that is to be used by the operator.

## 27.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

## 27.12.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 27.12.3. Parameters

| **EntitiesToStore** | |
|---|---|
| **Type** | static parameter |
| **Default** | 1 |
| **Range** | any |
| This parameter defines how many pixels, lines or frames have to be stored at maximum. The entity type is defined using parameter *EntityType*. | |

| **EntityType** | |
|---|---|
| **Type** | static parameter |
| **Default** | FRAME |
| **Range** | {FRAME, LINE, PIXEL} |
| This parameter defines the type of the entity to store. The parameter can buffer *EntitiesToStore* units of *EntityType*.<br><br>FRAME can only be chosen of the input link image protocol is VALT_IMAGE2D. LINE can be selected for protocols VALT_LINE2D and VALT_LINE1D, while PIXEL is always enabled. | |

| **ImplementationType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, BRAM, LUTRAM, URAM) |

Parameter *ImplementationType* influences the implementation strategy of the operator, i.e., which memory elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** The optimal implementation strategy is selected automatically based on the parametrization of the connected links.

**BRAM:** The operator uses the Block RAM of the FPGA.

**LUTRAM:** The operator uses the LUT RAM of the FPGA.
**URAM:** The operator uses the UltraRAM of the FPGA.



**Availability and Limitations of URAM**

The value *URAM* of the *ImplementationType* parameter is only available on the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms.

URAM has the following limitation:

- The *FifoDepth* of the operator must not exceed 4096. The *FifoDepth* depends on the parallelism, the *EntityType* parameter and the value of the *EntitiesToStore* parameter. If *FifoDepth* exceeds 4096, the *ImplementationType* parameter is in illegal state, see Section 3.7.1.5, 'Illegal Parameter Value States'.



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA resource runs short in a design, you can manually influence the FPGA resource

| ImplementationType | |
|---|---|
| | management using the values BRAM, LUTRAM, or URAM (URAM is only available for the imaFlex CXP-12 Quad and the imaFlex CXP-12 Penta platforms).<br><br>AUTO only selects between BRAM and LUTRAM. |

| FillLevel | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of buffer. The operator uses the FPGA resources efficiently. In some configurations it is possible to store more than 100% in the buffer. | |

| InfiniteSource | |
|---|---|
| **Type** | static parameter |
| **Default** | ENABLED |
| **Range** | {ENABLED, DISABLED} |
| This parameter activates support for infinite source operators like Camera operators. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

## 27.12.4. Examples of Use

The use of operator ImageFifo is shown in the following examples:

- Section 3.6.6, 'Timing Synchronization'

  Synchronization - Avoiding deadlocks.

- Section 9.3.1.4, 'Stitching of Two Cameras'

  Tutorial - Use of the operator to buffer one line for line duplication.

- Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche'

  Examples - Laroche Bayer De-Mosaicing filter

- Section 12.4.1.8, 'Modified Laroche Bayer De-Mosaicing Algorithm '

  Examples - Ressource Optimized Laroche Bayer De-Mosaicing filter

- Section 12.4.2.4, 'Color Plane Separation Option 4 - Sequential with Operator ImageBufferMultiRoI and a pre-sort of the Color Planes'

  Sequential DMA output of the color planes. The color separations is performed using operator ImageBufferMultiROI. An additional pre-sorting optimizes the bandwdith and resources.

- Section 12.6.2, 'Image Dimension Test'

  Example - The image dimension is measured and can be used to analyze the design flow.

- Section 12.6.7, 'Image Flow Control'

  Example - For debugging purposes of the designs internal data flow control in hardware and a possible compensation.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

- Section 12.9.9, 'Tap Geometry Sorting'

Examples - Scaling A Line Scan Image

- Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'

  Examples - High Dynamic Range According to Debevec

- Section 12.10.2, 'High Dynamic Range and Low Dynamic Range Example with a Weighted Linear Ansatz'

  Examples - High Dynamic Range with Linear Ansatz

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

# 27.13. Operator ImageSequence

**Operator Library: Memory**

| Availability for Hardware Platforms |
| --- |
| Please note that this operator is only available for target platforms of the microEnable 4 series (including PixelPlant). |

The main feature of this operator is that it buffers a sequence of *SequenceLength* successive input images. This sequence is output simultaneously. Moreover, a region-of-interest (ROI) feature is included.

This operator buffers the image stream in the *Frame Grabber RAM (DRAM)*. One VisualApplets resource of type *RAM* is required (see Section 3.8, 'Allocation of Device Resources').

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The simultaneous output of the image sequence is performed by using a kernel of *SequenceLength* columns. Each kernel component represents one image of the sequence i.e. corresponds to the image sequence number. For example at a sequence length of three, input image 0 will be output at kernel index 0, image 1 on kernel index 1, image 2 on kernel index 2. Next, the operator continuous with the next sequence, i.e. input image 3 will be output on kernel index 0, etc.

The operator does not change the bandwidth. It keeps the bandwidth constant for reading and writing. However the frame rate of the output images is 1/*SequenceLength* of the frame rate at the input. However, the output image is N times larger (due to the kernel) than a single input image.

The operator works like a FIFO buffer. As soon as enough images are buffered to generate a sequence, the operator will output the sequence. During this period, the next images are stored in the buffer.

Using parameters *XOffset*, *XLength*, *YOffset* and *YLength* the ROI size can be defined.



**No Filllevel and Flow Control**

The operator does not include parameters to check the fill level and no parameter to check for overflows. Moreover, no InfiniteSource parameter is included. The operator always assumes an infinite source at its input. It is not possible to check if the operator is in an overflow condition. See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information on flow control.

**Operator Restrictions**

- Empty frames are not supported.
- Images with varying line lengths are not supported.
- All input images of a sequence must have the same width and the same height. Otherwise the behavior of the operator is undefined.
- The minimum allowed input image width is XOffset + XLength and the minimum input image height is YOffset + YLength.
- The used RAM size is limited. Moreover, due to the internal implementation, the available buffer might be smaller than the physical space. The required memory is related to the maximum image width, maximum image height and the SequenceLength. If the maximum image width is a power of two value, the memory is efficiently used. Check Appendix A, '*Device Resources*' for more information.

## 27.13.1. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width.



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

## 27.13.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

## 27.13.3. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶❷ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any❸ | as I |
| **Kernel Columns** | 1 SequenceLength | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | 65535 | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷❸ The product of the bit width and the parallelism must not exceed the native ram data width. Check Appendix A, '*Device Resources*' for more information.

## 27.13.4. Parameters

| SequenceLength | |
|---|---|
| **Type** | static parameter |
| **Default** | 2 |
| **Range** | [2, possible with RAM size] |
| This parameter defines the number of images to be concatenated into a sequence. The value of parameter SequenceLength defines the number of columns of the kernel. | |

| XOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |

| XOffset | |
|---|---|
| **Range** | [0, Max.Img Width - XLength] |
| This parameter defines the x-coordinate of the upper left corner of the ROI.<br>The step size is the parallelism. | |

| XLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [2*Parallelism, Max.Img Width - XOffset] |
| This parameter defines the width of the ROI.<br>The step size is the parallelism. | |

| YOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Height - YLength] |
| This parameter defines the y-coordinate of the upper left corner of the ROI. | |

| YLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [2, Max.Img Height - YOffset] |
| This parameter defines the height of the ROI. | |

### 27.13.5. Examples of Use

The use of operator ImageSequence is shown in the following examples:

- Section 12.7.2, 'Noise Reduction'

  Examples - The average of two acquired images is calculated to reduce noise.

## 27.14. Operator KneeLUT

**Operator Library: Memory**

The operator implements an approximation of a look up table through a set of base points. Thus the name KneeLUT. The range of all representable values on the operator output is divided by Knee LUT into N intervals defined by the base points. The amount of intervals N is representable as a number of power of 2, i.e. N = 2^n, n is an integer number. Parameter *BasePointPows* defines n. The interval distance is constant and equal for all intervals.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

Each interval is described by a start base point and an end base point. The start base point defines the output value for the 1st interval step. The end base point defines the theoretical value for the end point of the interval that cannot be reached which is equal to the start base point of the next interval. All values inside the interval are approximated linearly. The end base point and the start base point of the consecutive interval have the same X-coordinate, i.e. there is no interval steps in-between.

Base points values for each interval in the KneeLUT can be defined individually. It is possible to define a discontinuous function over the entire output range. However, the sub functions inside each interval are linear with a gradient/slope defined through the start and the end base points. All start base points belong to the intervals. All end base points are excluded from the interval and can never be reached by the linear approximation.

All base points are normalized to the maximal representable value at the output of the operator, i.e. to $2^{OutputBitWidth} - 1$ . All start base points will be in the range from 0 to 1. Zero corresponds to the smallest representable value on the output. In this version of KneeLut it is equal to 0 because the operator supports only unsigned values. One represents the maximal representable value on the operator output, i.e. $2^{OutputBitWidth} - 1$ . All end base points can exceed the range of 0 to 1 to allow the last value included in the interval to be mapped to the highest and lowest numbers, i.e. to 0 and 1. This implies that the end base point might be larger than 1 or smaller than 0. Using the same scaling technique as for the start base points, the end points must be scaled to $2^{OutputBitWidth} - 1$ .

Let's have a look at two examples to illustrate the behavior of the operator.

1. Identity

   The first function of the KneeLUT is the identity. Suppose the following parameters of our KneeLUT:

   - BasePointPows = 2
   - Input Bit Width = 4
   - Output Bit Width = 5

   Thus we have to define four start base points and four end base points. For identity we define:

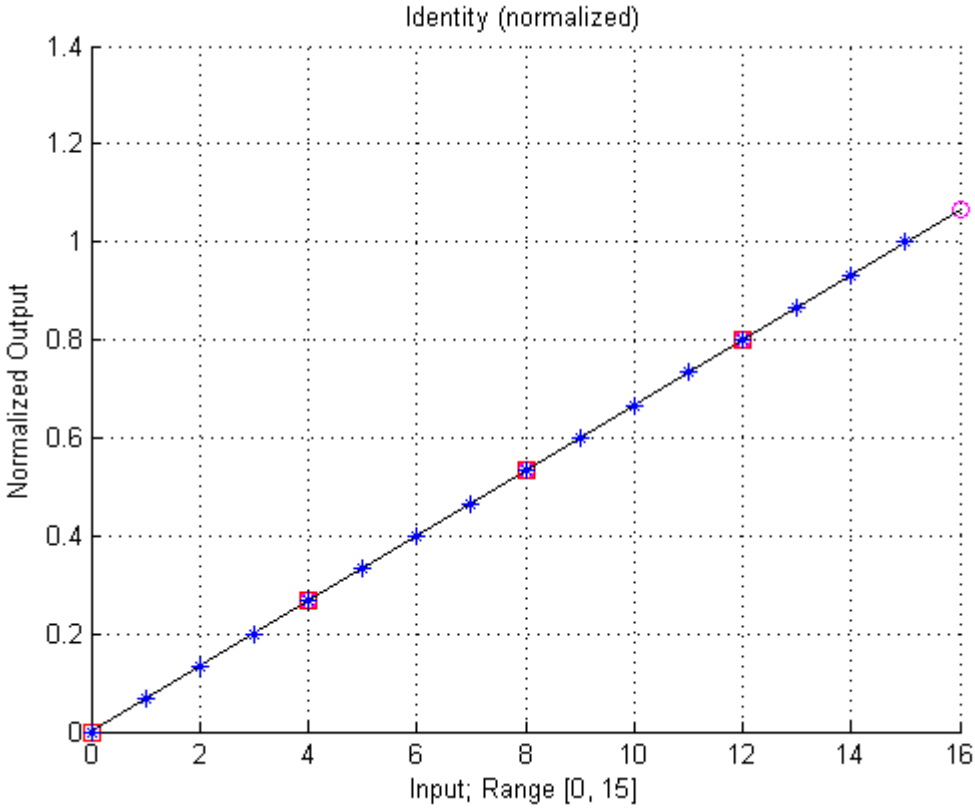
$$StartBasePoints = \{0 \quad 0.2667 \quad 0.566 \quad 0.8\}$$
$$EndBasePoints = \{0.2667 \quad 0.566 \quad 0.8 \quad 1.0667\}$$

   These values are determined by:

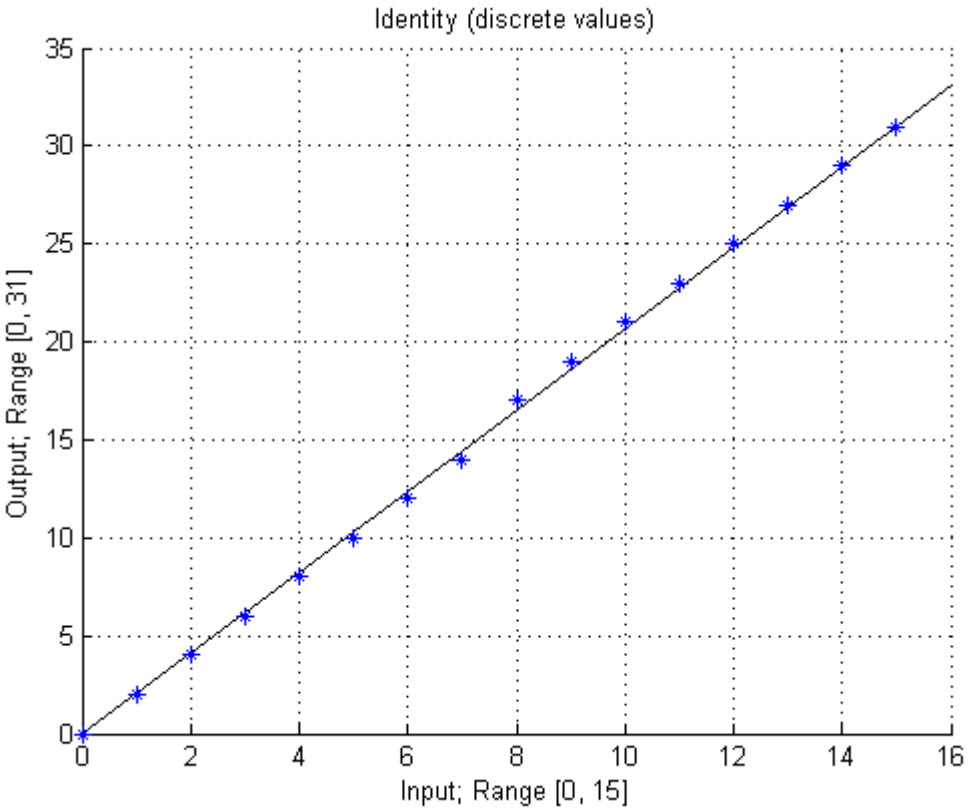
$$BasePoint(x) = x \times 4 \times \frac{1}{15} = x \times 2^{InputBitWidth - BasePointPows} \times \frac{1}{2^{OuputBitWidth} - 1}$$

   where x is the base point index.

   The following plot shows these base points. Start base points are marked with a red square and end base points are marked by a magenta circle. Moreover, a line is plot to illustrating the linear functions of the intervals.




The plot also shows the approximated values of the output (blue stars). Base points are normalized, i.e. a 0 is output value 0 and 1 is output value $2^{OutputBitWidth} - 1$ . The following plot shows the actual used lookup table with the given base points. As can be seen, the values are rounded to the next integer values.




The lookup table will calculate the following values:

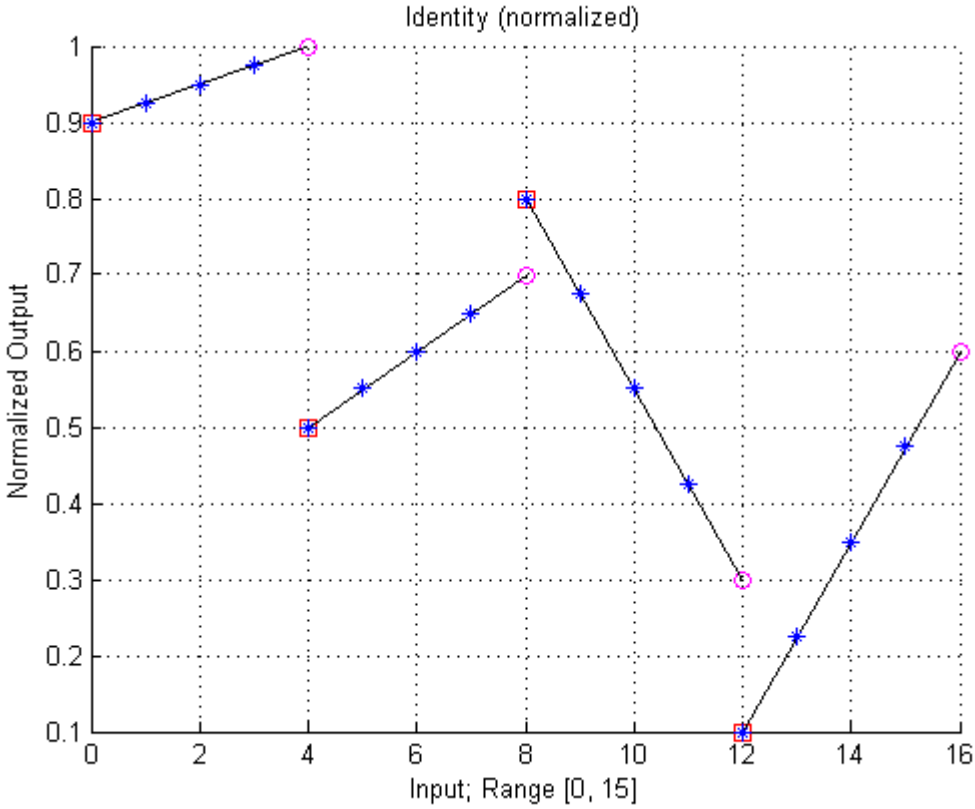
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 17 | 19 | 21 | 23 | 25 | 27 | 29 | 31 |

2. Discontinuous Function

   In the next example we take a look on a discontinous function. Let's assume the same KneeLUT parameters as give in the first example. This time, the base points are not defined by a function:

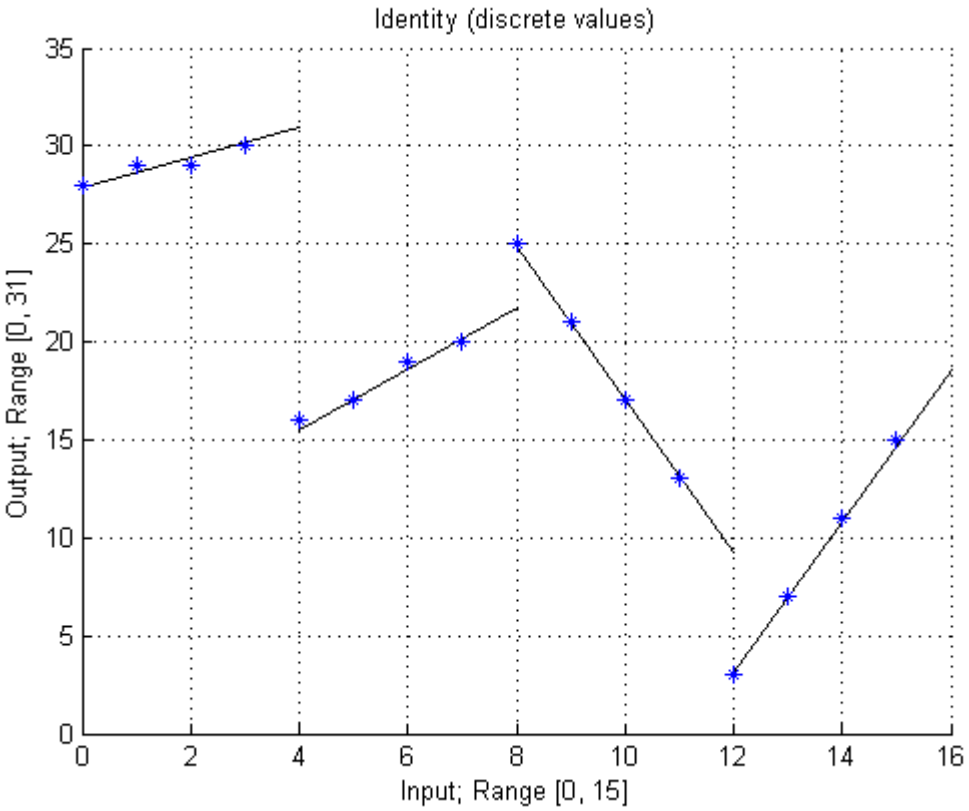
$$StartBasePoints = \{0.9 \quad 0.5 \quad 0.8 \quad 0.1\}$$
$$EndBasePoints = \{1 \quad 0.7 \quad 0.3 \quad 0.6\}$$

   Again, the following plot shows these base points. Start base points are marked with a red square and end base points are marked by a magenta circle. Moreover, a line is plot to illustrating the linear functions of the intervals.




The plot also shows the approximated values of the output (blue stars). Again, the plot is normalized to 1.

The following plot shows the actual used lookup table with the given base points. As can be seen, the values are rounded to the next integer values.




The lookup table will calculate the following values from the given base points:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 28 | 29 | 29 | 30 | 16 | 17 | 19 | 20 | 25 | 21 | 17 | 13 | 3 | 7 | 11 | 15 |

These values can be calculated by using the slope m(x) and the offset b(x) of each interval

$$m(x) = \frac{StartBasePoints\left(\left\lfloor \frac{x}{2^{InputBitWidth-BasePointPows}} \right\rfloor\right) - EndBasePoints\left(\left\lfloor \frac{x}{2^{InputBitWidth-BasePointPows}} \right\rfloor\right)}{2^{InputBitWidth-BasePointPows}}$$

$$b(x) = StartBasePoints\left(\left\lfloor \frac{x}{2^{InputBitWidth-BasePointPows}} \right\rfloor\right)$$

The result LUT(x) is then

$$LUT(x) = \left(\left(x \mod 2^{InputBitWidth-BasePointPows}\right) \times m(x) + b(x)\right) \times \left(2^{OutputBitWidth} - 1\right)$$

**Approximated Values out of Range**

If the approximated values are out of the available output range, the operator will not clip them to the maximum or minimum possible value. Values are undefined.

**Operator Restrictions**

- The following constraint is given:

$$InputBitWidth + OutputBitWidth - BasePointPows < 60$$

### 27.14.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

### 27.14.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | [1, 60] |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 27.14.3. Parameters

| **BasePointPows** | |
|---|---|
| **Type** | static parameter |
| **Default** | 6 |
| **Range** | [1, 60 - InputBitWidth + OutputBitWidth] |
| This parameter defines the number of base points for the KneeLUT according to the following formula: NumberOfBasePoints = 2 ^ BasePointsPows.<br><br>The step size is the parallelism. | |

| **StartBasePoints** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | identity function |
| **Range** | any |
| This field parameter defines a set of normalized 2^BasePointPows interval start base points. The points are normalized to the maximal representable value on the operator output.<br><br>Learn on how to configure field parameters in Section 3.7.1.3, 'Parameter Editing'. | |

| **EndBasePoints** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | identity function |
| **Range** | any |
| This parameter defines a set of normalized 2^BasePointPows interval end base points. The points are normalized to the maximal representable value on the operator output. The normalized values can exceed the range from 0 to 1 to allow the last interval point to reach the maximal respectively the minimal representable value. | |

### 27.14.4. Examples of Use

The use of operator KneeLUT is shown in the following examples:

- Section 12.11.3, 'Knee-Lookup Table 16 Bit'

  Examples - Shows the use of a lookup table for 16 Bit input and output data. For 16 bit a Knee LUT has to be used due to the limited block RAM resources.

# 27.15. Operator LineBuffer (imaFlex)

**Operator Library: Memory**

The *LineBuffer* operator buffers the image data stream in the Frame Grabber RAM (DRAM), which is a memory on the frame grabber but not in the FPGA. The *LineBuffer* operator is a replacement for the operator *ImageBuffer*, which is used for mE5 hardware platforms and provides the same features. imaFlex CXP-12 Quad and imaFlex CXP-12 Penta don't support the operator *ImageBuffer*, thus, when you convert a design from an mE5 hardware platform to an imaFlex CXP-12 Quad or imaFlex CXP-12 Penta platform, any instance of *ImageBuffer* is automatically replaced by an instance of *LineBuffer*.

One VisualApplets resource of type *RAM* is required. See Section 3.8, 'Allocation of Device Resources' for more information. Multiple resources of type *RAM* use the same physical RAM with the shared memory concept. Documentation for how to use the shared memory is available in section Section A.3, ' Shared Memory Concept '. Additionally, the *LineBuffer* operator features region-of-interest (ROI) support. The total number of bits (bit width times parallelism) must not exceed the memory limitations of the respective frame grabber.

The operator causes a latency of at least one line. Reading a line is started as soon as the line is fully written into the buffer. If the output is blocked, the operator buffers the input data.

The operator works like a FIFO. The buffer operates on image lines resulting in a very short memory latency. Any completed line of input data is immediately forwarded to the output as long as it is within the selected region of interest and the following modules can consume the data fast enough. As a result, frame data from a camera can be forwarded to the PC while the current frame transfer is still ongoing. The latency between ingoing and outgoing data increases, if the buffer gets filled with more lines. This happens, when the output is blocked for some time, in particular because the following modules can't process data at the same rate as the input bandwidth. Thus, the memory will only be filled, if the operator can't output the data at the same rate as the input provides data. Often, the operator is used to compensate peak input bandwidths.

With the parameters *XOffset*, *XLength*, *YOffset* and *YLength* you can define a region of interest. If the input image width is less than the sum of the *XOffset* and *XLength*, the *LineBuffer* operator still reads the parameterized *XLength*. In this case, the operator outputs undefined memory content for the additional pixels. If the input image height is less than the requested output image height, the operator only outputs the available lines. If the input image height is lower than *YOffset*

In *Line1D* application mode, the *YOffset* and *YLength* settings have no effect.

To measure the fill level of the buffer, the operator provides the following parameters: *FillLevel*, *LineCount*, and *Overflow*. *FillLevel* shows the fill level of the RAM in percent. *LineCount* provides the current number of lines that are saved in the buffer. The *Overflow* parameter is set to 1 when *FillLevel* is close to or equal to 100% and the next image to be stored in the buffer will exceed the RAM capacity. In case of an overflow, input data is discarded and the input image height is reduced. Overflows can only occur, if the data source is not stoppable, which means that *InfiniteSource* is set to *ENABLED*. As a result of an overflow, incomplete images might be stored in the memory. Incomplete images are automatically finalized and incoming data is discarded until there is space in the buffer for the next line.

The *InfiniteSource* parameter is used to specify whether the operator receives a data stream which can't be stopped, i.e. an unbuffered stream from a camera. Read Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information.

**Operator Restrictions**

- Empty frames are not supported.



**Available Memory Space**

The operator provides the read-only parameter *MaxLineCount* which reports the maximum number of lines (defined by *XLength*) that fit in the RAM. This number further depends on the presence of other memory operators in the current design and the data bit width.

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |

## 27.15.1. Bandwidth Optimization

In contrast to the *ImageBuffer* operator, the *LineBuffer* operator performs an automatic aggregation of pixels for matching the memory data width as close as possible, in order to reach an optimal performance.

If enabled by the *ParallelismConverter* parameter, you may adjust the output parallelism to a different value than the input parallelism. This enables defining an asymmetric peak data throughput between input and output of the operator. You can use this feature to avoid an additional instance of a *PARALLELdn* or *PARALLELup* operator when the following data processing pipeline needs to work with a different parallelism.

To enable the operator to handle long bursts of write data that exceed the available maximum bandwidth of the RAM, enable the *WritePriority* parameter. This is recommended with infinite data sources, such as cameras (*InfiniteSource* = *ENABLED*). When the available bandwidth of the RAM is exceeded due to a write burst, reading is inhibited. This allows temporarily more frequent write accesses to the RAM. If a camera delivers a burst of write data, it is expected that said burst is followed by a gap in the write data (otherwise the available RAM bandwidth is exceeded), which then can be used to temporarily ramp up the read bandwidth.

## 27.15.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, image data output |

## 27.15.3. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶❷ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any❷ | as I ❷ |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | 2^31-1❸ | as I |
| **Max. Img Height** | 2^31-1 | as I |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]

- For signed color inputs: [6, 63].

❷ The product of the bit width and the parallelism must not exceed the native RAM data width: *RamDataWidth*.

❸ If the *LineBuffer* operator converts the parallelism (*ParallelismConverter* = *Yes*), it automatically rounds the maximum image width at the output to the next multiple of *O.Parallelism*. If *O.Parallelism* is greater than *I.Parallelism*, *I.MaxImgWidth* must not be greater than 2^31-1-O.Parallelism so that the rounded maximum image width at the output doesn't exceed 2^31-1.

## 27.15.4. Parameters

| **RamDataWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of data bits that can be used. | |

| **XOffset** | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Width - XLength] |
| This parameter defines the x-coordinate of the upper left corner of the ROI.<br><br>The step size is the input parallelism. | |

| **XLength** | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 1024 |
| **Range** | [input parallelism, min(Max.Img Width, (2^RAM address width - 1) * internal RAM parallelism) - XOffset] |
| This parameter defines the width of the ROI.<br><br>The step size is the input parallelism. | |

| **YOffset** | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Height - YLength] |
| This parameter defines the y-coordinate of the upper left corner of the ROI. | |

| **YLength** | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 1024 |
| **Range** | [1, Max.Img Height - YOffset] |
| This parameter defines the height of the ROI. | |

| **ParallelismConverter** | |
|---|---|
| **Type** | static write parameter |
| **Default** | No |
| **Range** | {Yes, No} |
| This parameter defines whether the *LineBuffer* operator implements a parallelism converter. | |

| ParallelismConverter |
|---|
| • *No*: The operator doesn't perform a parallelism conversion. Input and output parallelism are always the same. When the input parallelism changes, the output parallelism is automatically changed to the new input parallelism value. The output parallelism can't be edited.<br>• *Yes*: The operator performs a parallelism conversion. The output parallelism is completely independent from the input parallelism and can be edited at will. |

| InfiniteSource | |
|---|---|
| **Type** | static write parameter |
| **Default** | DISABLED |
| **Range** | {ENABLED, DISABLED} |
| The *LineBuffer* operator can be inserted directly behind a camera operator. In this case, the *InfiniteSource* parameter must be set to *ENABLED*. The operator will then perform active overflow management and make sure that the operator can properly recover from overflows. The overflow can occur either when the data sink behind the operator stops or pauses the transmission and the buffer fill level reaches its maximum or when the input bandwidth is too high so the write data can't be transferred to the external RAM. When *InfiniteSource* is set to *DISABLED*, an inhibit signal is generated that stops the proceeding operator from transferring data, if the buffer fill level or input bandwidth get too high.<br><br>See Section 3.6.9, 'Infinite Sources / Connecting Cameras' for more information. | |

| WritePriority | |
|---|---|
| **Type** | static write parameter |
| **Default** | DISABLED |
| **Range** | {ENABLED, DISABLED} |
| The shared memory concept (Section A.3, ' Shared Memory Concept ') usually distributes the bandwidth equally amongst all connected memory operators (operators that use a resource of type *RAM*). If the *WritePriority* is *DISABLED*), the *LineBuffer* operator assigns the same priority to reading and writing. By setting *WritePriority* to *ENABLED*), the *LineBuffer* operator prioritizes writing over reading, but only while the temporary memory data rate is higher than the available bandwidth. The temporary prioritization of write data leads to a temporary slow down of the read process. Consequently, the average bandwidth must not exceed the available bandwidth for the *LineBuffer* operator. The write prioritization is recommended for any operator that is used with the *InfiniteSource* parameter set to *ENABLED*. When using the write prioritization with a stoppable source, make sure that the write bandwidth isn't constantly high, otherwise reading from the *LineBuffer* is stopped until the buffer is full. Since the write prioritization is a configuration for an individual operator, the effect of the write prioritization decreases with each additional memory operator in the design. | |

| FillLevel | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0%, 100%] |
| This parameter provides the fill level of DRAM in percent. | |

| MaxLineCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 2^RAM address width / (*XLength* / internal RAM parallelism + 1) |
| **Range** | [1, 2^RAM address width / 2] |
| This parameter provides the maximum number of lines that currently fit into the memory. The maximum number of lines that fit into the memory depends on the, RAM address width, which depends on the number of instantiated RAM operators. | |

| LineCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, *MaxLineCount*] |
| This parameter provides the current number of lines in the memory. | |

| Overflow | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 3] |
| This parameter indicates a buffer overflow. It's a 2-bit bitmap, where each bit indicates a different type of overflow. Bit 0 indicates a fill level overflow and bit 1 indicates a write bandwidth overflow. How long the *Overflow* parameter shows an overflow, depends on the *OverflowClearMode*. | |

| OverflowClearMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | AutoClear |
| **Range** | {AutoClear, ManualClear, ClearAfterRead, ClearWithProcessReset} |

*OverflowClearMode* determines how the *Overflow* parameter is cleared when the operator has recovered from an overflow. You can only reset the overflow status with this parameter, if the operator is not in overflow state anymore.

Clear modes:

- *AutoClear*: When the operator recovers from an overflow, the *Overflow* parameter is reset automatically.
- *ManualClear*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until it is manually reset by writing *ManualClear* into the *OverflowClearMode* parameter. In this mode, a process reset (e.g. acquisition stop) doesn't clear the *Overflow* parameter, which means the overflow is still visible after the acquisition has stopped.
- *ClearAfterRead*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until the *Overflow* parameter is read or a process reset occurs (e.g. when the acquisition is stopped).
- *ClearWithProcessReset*: When the operator recovers from an overflow, the *Overflow* parameter still shows the overflow until a reset occurs (e.g. when the acquisition is stopped).

# 27.16. Operator LineMemory

**Operator Library: Memory**

The LineMemory operator is a memory block which stores incoming images lines using random write access. The write addresses are specified with input link column address ColA. Thus for each input pixel value, a column address has to be specified.

After a line has been fully written to the memory, it is read and output using link O. The output image width is defined with parameters and is independent of the input image width. The memory is pre-initialized with values zero.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The memory size is defined by parameter *FrameWidth*.

The operator can be implemented in two variants. Namely, a single buffer implementation and a double buffer implementation. The first implementation saves RAM, but does not allow a timely overlap of writing and reading. Thus the input link I is stopped during the read state. The latter implementation doubles the RAM and allows to output an image line while accepting the next line at the input link I.

The operator has two states:

1. Write State: The value of Link I is stored at the address specified with ColA. This is done only if link WriteI is one. If link WriteI is zero, the current pixel is skipped. Writing to any address of the valid address range is possible. There is no necessity to write to each memory cell.

2. Read State: The memory is read out sequentially and produces a line of the parameterized width. By use of parameters XOffset and XLength it is possible to define the read address range i.e. a ROI. Reading the memory resets the content of the read addresses to zero. Thus if a memory cell is read which has not been written before, a zero will be output.

The toggling between these two states is triggered by an end-of-line at the input link I. Please note, that the line width at the output link is fixed to the size defined by parameters XOffset and XLength and is independent from the line width of the input link.

The operator can be useful for mirroring or sensor correction.

The operator uses the *FPGA-internal block RAM* or *UltraRAM memory* (URAM is only available for the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms). Thus, no VisualApplets frame grabber resources of type *RAM* are used. The FPGA-internal block RAM is limited. Full resolution frames might not fit into the FPGA-internal block RAM. Consider using operator *FrameBufferRandomRead* instead.

**Operator Restrictions**

- Empty frames are not supported.

  Images with varying line lengths are not supported.

## 27.16.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image data input<br>WriteI, write enable input<br>ColA, write column address for the pixel at I |
| **Output Link** | O, data input |

**Synchronous and Asynchronous Inputs**

- All inputs are synchronous to each other i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

## 27.16.2. Supported Link Format

| Link Parameter | Input Link I | Input Link WriteI |
|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Input Link ColA | Output Link O |
|---|---|---|
| **Bit Width** | auto❷ | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | as I | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | as I | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | as I | parameter FrameWidth |
| **Max. Img Height** | as I | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ The bit width of the column address is:

$$ColABitWidth = \lceil log_2(FrameWidth) \rceil$$

## 27.16.3. Parameters

| ImplementationType (imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms) | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, BRAM, URAM) |

Parameter *ImplementationType* influences the implementation strategy of the operator, i.e., which memory elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** The optimal implementation strategy is selected automatically based on the parametrization of the connected links.

**BRAM:** The operator uses the Block RAM of the FPGA.

**URAM:** The operator uses the UltraRAM of the FPGA.

| ImplementationType (imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms) |
|---|



**Availability and Limitations of URAM**

The *ImplementationType* parameter and thus the option of using URAM is only available on the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms.

URAM has the following limitation:

- The *FrameWidth* parameter of the operator must not exceed 4096.



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA resource runs short in a design, you can manually influence the FPGA resource management using the values BRAM, or URAM.

| Implementation | |
|---|---|
| **Type** | static parameter |
| **Default** | SingleBuffer |
| **Range** | {SingleBuffer, DoubleBuffer} |
| This parameter selects the implementation of the LineMemory (see above). | |

| FrameWidth | |
|---|---|
| **Type** | static parameter |
| **Default** | 1024 |
| **Range** | [0, 65534] |
| This parameter selects the image width of the output link in pixels. | |

| XOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, Max.Img Height - XLength] |
| This parameter defines the x-coordinate of the upper left corner of the ROI. | |

| XLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [1, Max.Img Width - XOffset] |
| This parameter defines the width of the ROI. | |

## 27.16.4. Examples of Use

The use of operator LineMemory is shown in the following examples:

- Section 12.9.6, 'Line Mirror'

  Examples - Shows how to vertically mirror an image. Note the mirroring of the parallel words and the pixel.

- Section 13.4, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

# 27.17. Operator LineMemoryRandomRd

**Operator Library: Memory**

The LineMemoryRandomRd (line memory random read) operator is a small memory block with random read access. The random read of the data can be performed by transferring addresses to port RColA (read column address). The operator will use the addresses to read the line data by the given coordinates. The resulting output frame will have the image width of the address input.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The LineMemoryRandomRd operator stores incoming image lines and allows random read access. The line data is transferred into the operator via link I.

The required memory size is defined by the image width of the input frame (Max. Img Width).

The operator can be implemented in two variants. Namely, a single buffer implementation and a double buffer implementation. The first implementation saves RAM, but does not allow a timely overlap of writing and reading. Thus the address input links RColA and RRowA are stopped during the write state. The latter implementation doubles the RAM and allows to random read the last frame while accepting the next frame at the input link I.

The operator has two states:

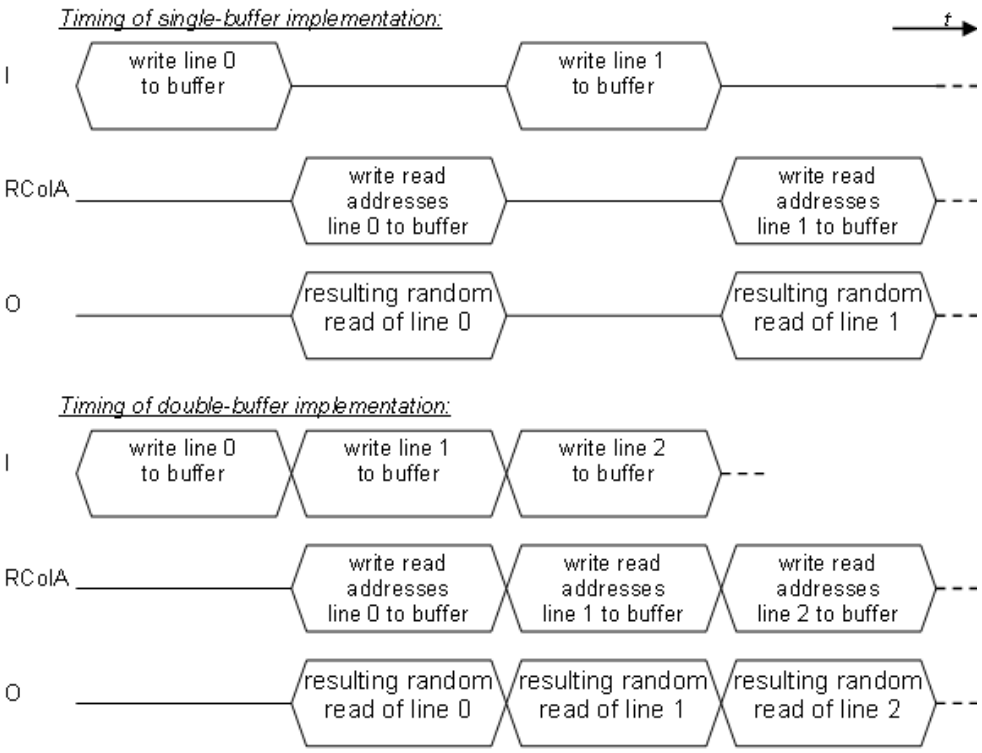
1. Write State: The pixels of link I are stored linear in the buffer i.e. the coordinates of the pixels form the addresses in the buffer.

2. Read State: The memory is read-out using the addresses given at the input RColA. The number of addresses and address link image width define the image output width. For example, if a line consisting of only one pixel is input at the address inputs, the output frame width will only consist of one pixel. The other pixels of the input lines are discarded.

The image data input and the address input are not synchronous to each other. They may have different image dimensions.

Please not the timing of the input links. The address input must not be sourced by the same operator as the data link input I without buffering. This is because while writing the image data into the operator, no addresses to read the current frame can be accepted. Only if the frame is fully stored into the buffer, addresses can be accepted. In many applications operator  is used to generate the images for the addresses.

The timing is visualized in the following figure.




The operator uses *FPGA-internal block RAM* memory. Thus non VisualApplets frame grabber resources of type *RAM* are used. The FPGA-internal block RAM is limited. Full resolution frames might not fit into the FPGA-internal block RAM. Consider using operator *FrameBufferRandomRead* instead.

**Operator Restrictions**

- Empty frames are not supported.

  Images with varying line lengths are not supported.

## 27.17.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image data input<br>RColA, read column address for the pixel at I |
| **Output Link** | O, data input |

**Synchronous and Asynchronous Inputs**

- All inputs are asynchronous to each other.

## 27.17.2. Supported Link Format

| Link Parameter | Input Link I | Input Link RColA | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | auto❷ | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | 1 | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |

| Link Parameter | Input Link I | Input Link RColA | Output Link O |
|---|---|---|---|
| **Img Protocol** | VALT_IMAGE2D | as I | as I |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | any | as RColA |
| **Max. Img Height** | any | as I | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The bit width of the row address is:

$$RowABitWidth = \lceil log_2(InputMax.ImageWidth) \rceil$$

## 27.17.3. Parameters

| Implementation | |
|---|---|
| **Type** | static parameter |
| **Default** | SingleBuffer |
| **Range** | {SingleBuffer, DoubleBuffer} |
| This parameter selects the implementation of the FrameMemoryRandomRd (see above). | |

## 27.17.4. Examples of Use

The use of operator LineMemoryRandomRd is shown in the following examples:

- Section 13.4, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

# 27.18. Operator LUT

**Operator Library: Memory**

The operator LUT is a lookup table of dynamic content. The values of the input link define the addresses of the LUT. The output link will then provides the value stored at this address. The input bit width defines the number of addresses in the LUT. The output bit width defines the value range of each LUT element. Both can be set to any value. The LUT content can be changed before synthesis or dynamic during runtime.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

## 27.18.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

## 27.18.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 16]❶ | [1, 63 unsigned / 64 signed]❷ |
| **Arithmetic** | {unsigned} | {63 bit unsigned, 64 bit signed} |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ If you have set parameter *ImplementationType* to *URAM*, the *Bit Width* of the input link must be set to a value between 1 and 14.

URAM is only available for the imaFlex CXP-12 Quad and the imaFlex CXP-12 Penta platforms.

❷ If you have set parameter *ImplementationType* to *URAM*, the *Bit Width* of the output link depends on the bit width of the input link:

- If the bit width of the input link <= 12, you can set any value.
- If the bit width of the input link = 13 or 14, set the bit width of the output link to a value between 1 and 16.

URAM is only available for the imaFlex CXP-12 Quad and the imaFlex CXP-12 Penta platforms.

## 27.18.3. Parameters

| LUTcontent | |
|---|---|
| **Type** | dynamic/static read/write parameter |

| LUTcontent | |
|---|---|
| **Default** | identity function |
| **Range** | [0, 2^OuptuBitWidth-1] |

This field parameter defines the LUT content. The number of field values is defined by the operator's input bit width.

Learn on how to configure field parameters in Section 3.7.1.3, 'Parameter Editing'.



**Input Bit Width Changed**

If the input bit width is changed, the number of field elements in this parameter changes. If the input bit width is reduced, values are truncated and deleted. If the input bit width is increased, the new field elements are set to 0.



**Warning Status**

If the BRAM is operated at less than 25% capacity, the *LUTcontent* parameter can go into warning status.

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, BRAM, LUTRAM, URAM) |

Parameter *ImplementationType* influences the implementation strategy of the operator, i.e., which memory elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** The optimal implementation strategy is selected automatically based on the parametrization of the connected links. AUTO only selects between BRAM and LUTRAM.

**BRAM:** The operator uses the Block RAM of the FPGA.

**LUTRAM:** The operator uses the LUT RAM of the FPGA.

**URAM:** The operator uses the UltraRAM of the FPGA.



**Availability and Limitations of URAM**

The value *URAM* is only available on the imaFlex CXP-12 Quad and the imaFlex CXP-12 Penta platforms.

URAM has the following limitations:

- The *Bit Width* of the input link must be set to values between 1 and 14.
- The *Bit Width* of the output link depends on the bit width of the input link:
  - If the bit width of the input link <= 12, you can set any value.
  - If the bit width of the input link = 13 or 14, set the bit width of the output link to a value between 1 and 16.



**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA resource runs short in a design, you can manually influence the FPGA resource

| ImplementationType | |
|---|---|
| | management using the values BRAM, LUTRAM, or URAM (URAM is only available for the imaFlex CXP-12 Quad and the imaFlex CXP-12 Penta platforms).<br><br>AUTO only selects between BRAM and LUTRAM. |

## 27.18.4. Examples of Use

The use of operator LUT is shown in the following examples:

- Section 12.4.3, 'HSL Color Classification'

  Examples - Color Classification is very simple on HSL images. The applet converts the RGB image into an HSL image and performs a color classification. The hue is filtered using a lookup table. Moreover, the saturation and lightness is thresholded using custom threshold values.

- Section 12.9.3.2.4, 'Geometric Transformation and Distortion Correction'

  Examples- Geometric Transformation and Distortion Correction using **PixelReplicator**

- Section 12.9.3.2.5, 'Distortion Correction'

  Examples- Distortion Correction

- Section 12.9.8, 'Scaling a Line Scan Image'

  Examples - Scaling A Line Scan Image

- Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'

  Examples - High Dynamic Range According to Debevec

- Section 12.10.2, 'High Dynamic Range and Low Dynamic Range Example with a Weighted Linear Ansatz'

  Examples - High Dynamic Range with Linear Ansatz

- Section 12.11.1, 'Lookup Table 8 Bit'

  Examples - Shows the use of a 8 Bit to 8 Bit lookup table.

- Section 12.11.2, 'Lookup Table 10 to 16 Bit'

  Examples - Shows the use of a lookup table with 10 bit input and 16 bit output.

- Section 12.11.4, 'Knee-Lookup Table 24 Bit Color'

  Examples - In this example three lookup tables are used for RGB color correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

## 27.19. Operator RamLUT

**Operator Library: Memory**

This operator implements a large Look Up Table (LUT) based on the *Frame Grabber RAM*, usually DRAM. One VisualApplets resource of type *RAM* is required (see Section 3.8, 'Allocation of Device Resources').

Operator *RamLUT* supports kernels on the output.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.

The typical application fields of this operator are color space classification problems. For example on a microEnable IV family board the operator allows a lookup table of 2^24 entries, i.e. a classification look up table can be defined containing a value for each of the colors in the RGB 24 color space. The operator facilitates the programming of the LUT content in two ways. For both ways, the LUT has to be programmed post to the synthesis process at the time when the applet is used during runtime.

1. One possibility of programming the LUT is a register interface which offers address- value-parameters *InitAddress* and *InitData*. First, set the address using parameter *InitAddress*. Next, write the data into parameter *InitData*. Writing the data causes the operator to actually write to the LUT and replace the previous value. If you use kernels, writing to the last kernel element will cause the operator to actually write to the LUT and replace the previous value.

2. The second possibility of programming the LUT is the usage of a file containing the content. The file access is faster than the register access, but requires some attention.

   The file can be in different formats. The format has to be defined by parameter *InitFileMode*:

   - **`text_with_checks`**

     In this mode the init file has to be a text file where the entries are separated by either blanks, tab stops, line feeds (LF) or carriage return line feeds (CRLF). The operator checks this files for errors and reports an error if the file cannot be used.

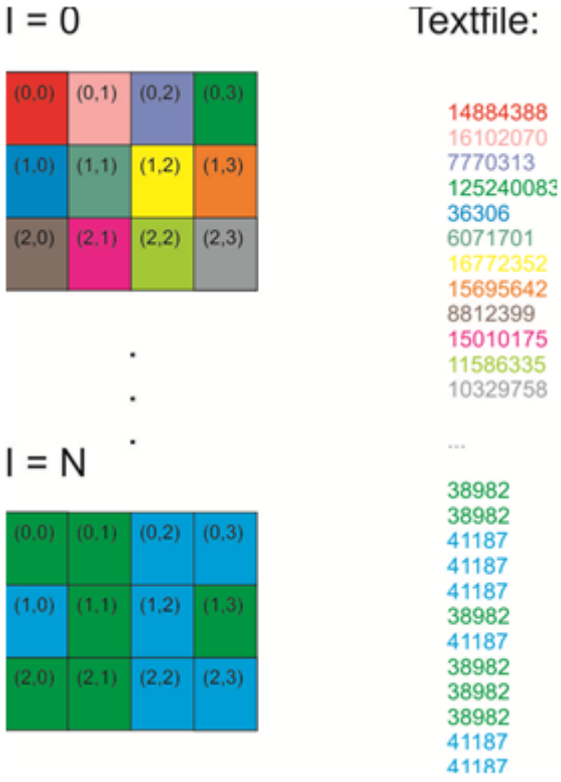
     Each Entry represents a kernel element and needs to be a decimal number within the correct range.

     The following figure demonstrates the file for a 4x3 kernel:



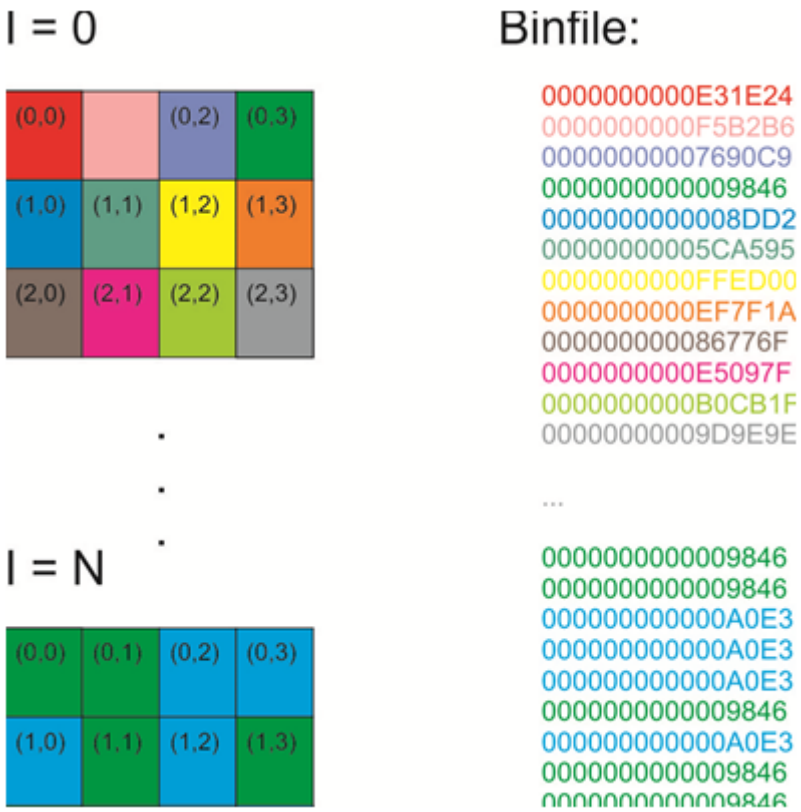

- **text_raw**

  The mode is similar to mode **text_with_checks** but comprises less checks for errors. In this mode each entry must be in a separate line. Loading a file in this mode is faster compared to the previous mode. The Example from above would look like that in text_raw mode:



- **binary**

  The binary mode assumes a binary file, where 8 bytes are used for each kernel entry of a LUT element. If the Kernel entry is smaller than 64 Bit, the unused Bits will be ignored. This is the fastest method of writing values into the LUT. The Example from above would look like that in binarymode:




  In all three modes the number of entries in the files has to match with the number of LUT elements. Thus, the files must contain 2^InputBitWidth entries. A partial initialization is not possible.

  Parameter *InitFilename* specifies the file containing the initial values. Finally, writing value one to parameter *LoadInitFile* starts the reading of the file and if accepted writing of the values to the hardware. Writing 0 to parameter *LoadInitFile* will not cause the loading of the values. This can be useful if it is unwanted that the initial file is loaded to the hardware during the applet initialization process.

Note that the DRAM technology offers very poor performance for random access. In a worst case scenario when each consecutive access to the DRAM causes a page miss and requires a new row activation, the DRAM throughput will suffer 90% loss of the maximal performance, resulting in about 10% bandwidth usage. However, by combining several DRAM banks together it is possible to increase the throughput even for the worst case. Tie 2 or more RamLUT operators together and let each of them process a different lookup job. If successive input values to not change quickly, the performance of the operator is much higher. Check the respective RAM technology and dimensions of your frame grabber in the hardware user guides.

### 27.19.1. Bandwidth Optimization

The theoretical bandwidth [bits/second] going through an operator that uses the *Frame Grabber RAM (DRAM)* is calculated in accord with the following formula:

$$TheoreticalBandwidth = SystemClock[inHz] \times BitWidth \times Parallelism$$

However, the actual bandwidth is always less than the theoretical bandwidth due to the DRAM efficiency.

The **maximum bandwidth** going through the operator is reached if the product of Bit Width and Parallelism is equal to the internal RAM Port Width x 2 (true for read-only parameters).



**Platform-specific values**

RAM Port Width and System Clock are platform-specific. See Appendix A, '*Device Resources*' for detailed information on your individual platform.

## 27.19.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

## 27.19.3. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, RAM Address Width]❶ | [1, Ram Data Width]❷ |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | [1, Ram Data Width / Bit Width]❷ |
| **Kernel Rows** | 1 | [1, Ram Data Width / Bit Width / Kernel Columns]❷ |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The allowed input bit width depends on the physical memory of the frame grabber. Check Appendix A, '*Device Resources*' for more information.

❷ The allowed output bit width depends on the physical memory of the frame grabber. The Bit Width is limited to 64 Bit, even if *RamDataWidth* is bigger. Check Appendix A, '*Device Resources*' for more information. The product of *O.BitWidth*, *O.Kernel Columns* and *O.Kernel Rows* must be lower than *RamDataWidth*.

## 27.19.4. Parameters

| **RamDataWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of data bits that can be used at the RAM output interface. | |

| **RamAddressWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |

| RamAddressWidth | |
|---|---|
| **Range** | Integer number |
| This parameter provides the number of RAM address bits available. | |

| InitAddress | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^InputBitWidth-1] |
| This parameter defines the address of the data defined by parameter *InitData*. See descriptions above. | |

| InitData | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^OutputBitWidth-1] |
| This parameter defines the data which will be written to the address defined by parameter *InitAddress*. Writing to this parameter starts the actual writing into hardware. See descriptions above. | |

| InitFileLoadMode | |
|---|---|
| **Type** | static parameter |
| **Default** | text_with_checks |
| **Range** | {text_with_checks, text_raw, binary} |
| The parameter defines the file format and the mode of the file being load to the lookup table. See descriptions above. | |

| InitFileName | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | InitRamLut.txt |
| **Range** | |
| This parameter defines the name of the initialization file. | |

| LoadInitFile | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| Writing value one to this parameters starts the loading the file specified by parameter *InitFileName* into the LUT. See descriptions above. | |

## 27.19.5. Examples of Use

The use of operator RamLUT is shown in the following examples:

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

- Section 13.8, 'Functional Example for Specific Operators of Library Color, Base and Memory'

  Examples - Demonstration of how to use the operator

# 27.20. Operator RamLUT (imaFlex)

**Operator Library: Memory**

This operator implements a large lookup table (LUT) based on the *Frame Grabber RAM*, usually DRAM. One VisualApplets resource of type *RAM* is required (see Section 3.8, 'Allocation of Device Resources'). Multiple resources of type *RAM* use the same physical RAM with the shared memory concept. Documentation for how to use the shared memory is available in the Application Note: Shared Memory [https://docs.baslerweb.com/visualapplets/application-note-shared-memory].

The operator *RamLUT* supports kernels on the output.

Typical use cases for this operator are color space classification problems.

The operator facilitates the programming of the LUT content in two ways. For both ways, the LUT has to be programmed after the synthesis process at the time when the applet is used during runtime.

1. One possibility of programming the LUT is a register interface which offers address- value-parameters *InitAddress* and *InitData*. First, set the address using parameter *InitAddress*. Next, write the data into parameter *InitData*. Writing the data causes the operator to actually write to the LUT and replace the previous value. If you use kernels, writing to the last kernel element causes the operator to actually write to the LUT and replace the previous value.
2. The second possibility of programming the LUT is the usage of a file containing the content. The file access is faster than the register access, but requires some attention. The file can be in different formats. The format has to be defined by parameter *InitFileMode*:

   - **`text_with_checks`**

     In this mode the init file has to be a text file where the entries are separated by either blanks, tab stops, line feeds (LF), or carriage return line feeds (CRLF). The operator checks these files for errors and reports an error if the file can't be used.

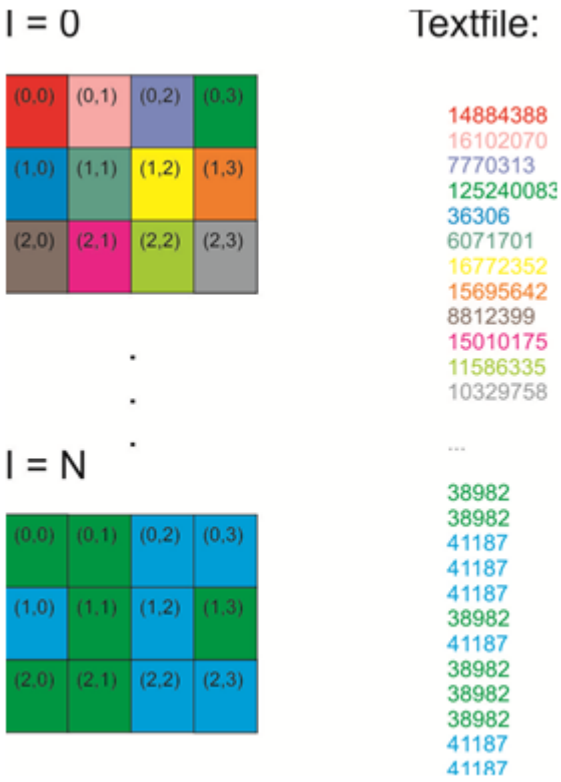
     Each entry represents a kernel element and needs to be a decimal number within the correct range.

     The following figure demonstrates the file for a 4x3 kernel:



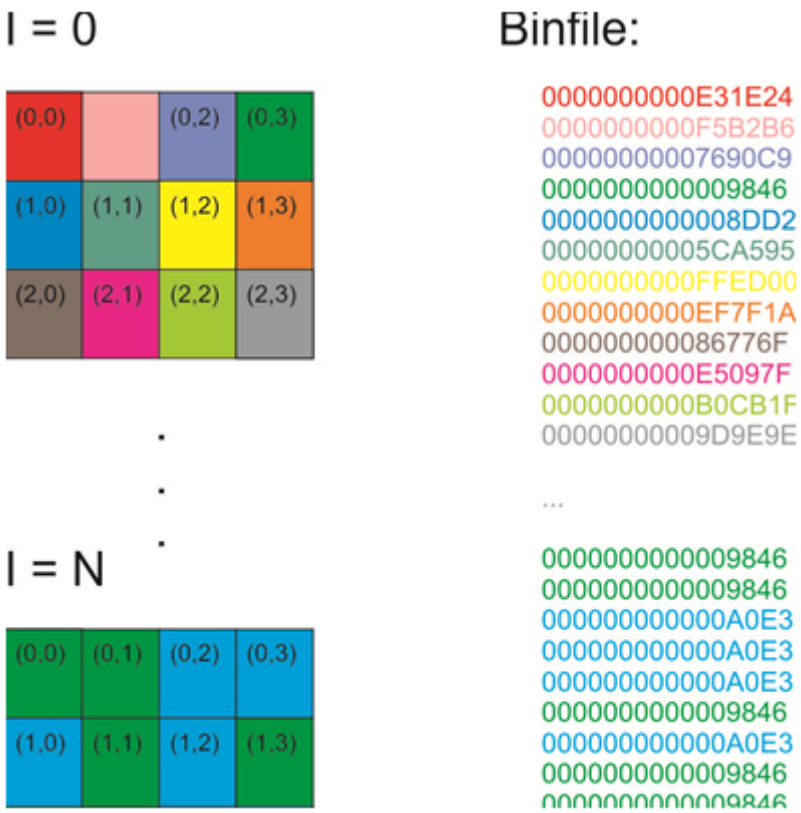
- **text_raw**

  This mode is similar to mode **text_with_checks** but comprises less checks for errors. In this mode each entry must be in a separate line. Loading a file in this mode is faster compared to the **text_with_checks** mode. The Example from above would look like that in **text_raw** mode:




- **binary**

  The **binary** mode assumes a binary file, where 8 bytes are used for each kernel entry of a LUT element. If the kernel entry is smaller than 64 Bit, the unused bits are ignored. This is the fastest method of writing values into the LUT. The example from above would look like that in **binary** mode:




In all three modes the number of entries in the files has to match the number of LUT elements. Thus, the files must contain 2^InputBitWidth kernel entries. A partial initialization is not possible.

Parameter *InitFilename* specifies the file that contains the initial values. Finally, writing value **one** to parameter *LoadInitFile* starts reading the file and, if accepted, writing the values to the hardware. Writing **0** to parameter *LoadInitFile* doesn't cause the loading of the values. This can be useful if you do not want the initial file to be loaded to the hardware during the applet initialization process.

Note that the DRAM technology offers very poor performance for random access. In a worst case scenario when each consecutive access to the DRAM causes a page miss and requires a new row activation, the DRAM throughput suffers 90% loss of the maximal performance, resulting in about 10% bandwidth usage.

The input link can handle dynamic line widths, as well as empty lines and empty frames. Consequently, the output link can handle the same image dimensions.

### 27.20.1. Bandwidth Optimization

For optimal performance, the used number of data bits should match as closely as possible the number provided in the module parameter *RamDataWidth*. The maximum bandwidth going through the operator is reached if the product of bit width and kernel size is equal to the internal RAM port width *RamDataWidth*.

| **Available for Hardware Platform** |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |

## 27.20.2. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, read address input |
| **Output Link** | O, image data output |

## 27.20.3. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, RAM Address Width]❶ | [1, 64]❷ |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | [1, Ram Data Width / Bit Width] |
| **Kernel Rows** | 1 | [1, Ram Data Width / Bit Width / Kernel Columns] |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The allowed input bit width depends on the physical memory of the frame grabber and on how many operators share access to the RAM. Parameter *RamAddressWidth* provides the maximum address bits which can be used. Note that this number may change depending on the number of RAM-based operators in the design.

❷ The allowed output bit width depends on the physical memory of the frame grabber. The bit width is limited to 64 bit but several kernel components may be stored. The product of (bit width * kernel size) must not exceed the native RAM data width shown in parameter *RamDataWidth*.

## 27.20.4. Parameters

| **RamDataWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of data bits that can be used at the RAM interface. It's the maximum number of bits the output can provide (if kernels are used properly). | |

| **RamAddressWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | Integer number |
| This parameter provides the number of address bits that can be used. | |

| **InitAddress** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^InputBitWidth-1] |

| **InitAddress** |
| --- |
| This parameter defines the address of the data defined by the parameter *InitData*. See the descriptions above. |

| **InitData** | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^OutputBitWidth-1] |
| This parameter defines the data that is written to the address defined by the parameter *InitAddress*. Writing to this parameter starts the actual writing into hardware. See the descriptions above. | |

| **InitFileLoadMode** | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | text_with_checks |
| **Range** | {text_with_checks, text_raw, binary} |
| This parameter defines the file format and the mode of the file that is loaded into the lookup table. See the descriptions above. | |

| **InitFileName** | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | InitRamLut.txt |
| **Range** | |
| This parameter defines the name of the initialization file. | |

| **LoadInitFile** | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| Writing value **one** to this parameter starts loading the file specified by the parameter *InitFileName* into the LUT. See the descriptions above. | |

# 27.21. Operator ROM

**Operator Library: Memory**

The operator ROM is a lookup table of dynamic content. The values of the input link define the addresses of the LUT. The output link provides the value stored at this address. The input bit width defines the number of addresses in the LUT. The output bit width defines the value range of each LUT element. Both can be set to any value. The LUT content can be changed before the build, or dynamically during runtime.

For information on the latency of the operator, see Table 27.2, 'Individual Latencies of the Operators in Library Memory'.



**Same as Operator LUT**

The ROM operator is identical to operator *LUT*.

## 27.21.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, data input |

## 27.21.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 16] | [1, 63 unsigned/ 64 signed] |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 27.21.3. Parameters

| ROMcontent | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | identity function |
| **Range** | [0, 2^OutputBitWidth-1] |
| This field parameter defines the LUT content. The number of field values is defined by the operator's input bit width.<br><br>Learn on how to configure field parameters in Section 3.7.1.3, 'Parameter Editing'. | |

| ROMcontent |
|---|
| **Input Bit Width Changed**<br><br>If the input bit width is changed, the number of field elements in this parameter changes. If the input bit width is reduced, values are truncated and deleted. If the input bit width is increased, the new field elements are set to 0. |

| ImplementationType | |
|---|---|
| **Type** | static write parameter |
| **Default** | AUTO |
| **Range** | (AUTO, BRAM, LUTRAM) |

Parameter ImplementationType influences the implementation strategy of the operator, i.e., which memory elements are used for implementing the operator.

You can select one of the following values:

**AUTO:** The optimal implementation strategy is selected automatically based on the parametrization of the connected links.

**BRAM:** The operator uses the Block RAM of the FPGA.

**LUTRAM:** The operator uses the LUT RAM of the FPGA.

**Use AUTO in General**

Normally, the parameter should be set to AUTO. In special cases, i.e., if one kind of FPGA ressource runs short in a design, you can manually influence the FPGA resource management using the values BRAM or LUTRAM.

## 27.21.4. Examples of Use

The use of operator ROM is shown in the following examples:

- Section 12.11.1, 'Lookup Table 8 Bit'

  Examples - Shows the use of a 8 Bit to 8 Bit lookup table.

- Section 12.11.2, 'Lookup Table 10 to 16 Bit'

  Examples - Shows the use of a lookup table with 10 bit input and 16 bit output.

- Section 12.11.4, 'Knee-Lookup Table 24 Bit Color'

  Examples - In this example three lookup tables are used for RGB color correction.

# 28. Library Parameters




Library *Parameters* provides several major advantages for the parametrization of designs during design time AND during runtime:

- It allows to set paths to specific parameters which may be deeply embedded in hierarchical structures. This way, even parameters in protected hierarchical boxes (or in protected hierarchical boxes that are nestled in other protected hierarchical boxes) can be set and reset during runtime, since their parametrization interface is lifted up in the hierarchy to a hierarchical level that is not access restricted.
- You can set up nets and/or chains of parameters in your design, allowing to forward one parameter value to various other operators. This way, you can simplify the parametrization of the final applet: The final user may, e.g., set one parameter once, and, triggered from this write process, various other parameters within the design will be set accordantly without any further user interaction.
- With the translator operators of the library, you can not only forward parameter values, but make the design calculate new values out of the current values of various parameters and write the result to still other parameters any place in the design. You define the formulas yourself. The syntax of the formulas complies to the GenICam standard (GenICam API version 2.0) and is described below.

The good thing about the parameters library is that you can set up connections (nets or chains) between parameter values in your design without actually touching the data flow structure of the design.

Library *Parameters* contains seven reference operators and four translation operators. Find more information about both types of operators below.

You find example designs using the operators of the Parameters Library in your VisualApplets installation: **<installation_directory>\VisualApplets\Examples\AdvancedVAFunctions\Parameters Library**.



**Availability**

To use the **Parameters** library, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

**********

**The Reference Operators**

The *Reference* operators of the *Parameters* library allow a 1:1 mapping of an operator parameter to the map parameter *Value* (or *Field*) of the reference operator. You specify the referenced parameter (path and name) in parameter "Reference".

You can define a target within the design hierarchy where the referenced parameter will be visible and accessible. The target is defined by two parameters: "DisplayHierarchy" and "DisplayName". In "DisplayHierarchy" you address a hierarchical box within the same process the reference operator itself is located in. In parameter "DisplayName" you set up the name for the target parameter. If "DisplayHierarchy" and "DisplayName" are both set, the parameter defined by "DisplayName" will be available in the hierarchical box addressed by "DisplayHierarchy". If "DisplayName" is empty, no parameter is created. If "DisplayName" is set, but "DisplayHierarchy" is empty, the parameter defined by "DisplayName" is added to the parent hierarchy (i.e., the hierarchical box the reference operator itself is located in).

This allows you to make a parameter of your design available on any hierarchical level (of the design) you want. For example, you can make the parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a reference operator, or reference the parameter you have defined in the parameters "DisplayName" and "DisplayHierarchy". This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



**All connected parameters update at once**

If one of the connected parameters is changed (referenced parameter, map parameter "Value", or target parameter), the value is changed in all other connected parameters, too.

To set up map parameters, you simply proceed 2 to 3 steps:

1. Instantiate a reference operator.

2. Define a source parameter (e.g., a parameter that is deeply nestled within the hierarchy). Immediately, the value of the source parameter is mirrored in the map parameter of the reference operator instance.

3. (Optionally) define name and location (path to target hierarchical box and parameter name) of a second map parameter.

All map parameters are treated as normal parameters by the environment. You can access them via the Framegrabber API or the Framegrabber SDK tools (microDiagnostics and microDisplay) the same way you access "original" operator parameters.

Each instance of a reference operator allows to set up an 1:1 relation between the value of any parameter in your design and the map parameter (parameter *Value*) of the reference operator instance.

**Data types supported by reference operators:**

To create map parameters, you first need to instantiate one reference operator. For each possible data type, you have one reference operator available:

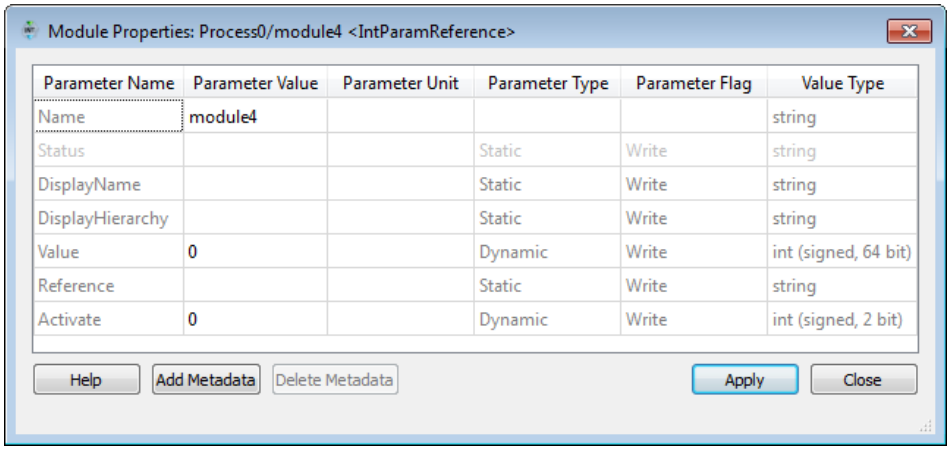
| Operator Name | Data types that are supported by this operator |
|---|---|
| IntParamReference | Integer (VA_SINT, VA_UINT) |
| EnumParamReference | Enumeration (VA_ENUM) |
| FloatParamReference | Floating Point (VA_DOUBLE) |
| IntFieldParamReference | Integer Fields (VA_SINTFIELD, VA_UINTFIELD) |
| FloatFieldParamReference | Floating Point Fields (VA_DOUBLEFIELD) |
| StringParamReference | String (VA_STRING,VA_FILENAME,VA_METADATA) |
| ResourceReference | Integer (VA_SINT, VA_UINT) |

Table 28.1. Data types supported by reference operators

**Procedure for creating map values with Reference Operators:**

1. Instantiate the reference operator of the data type you need.

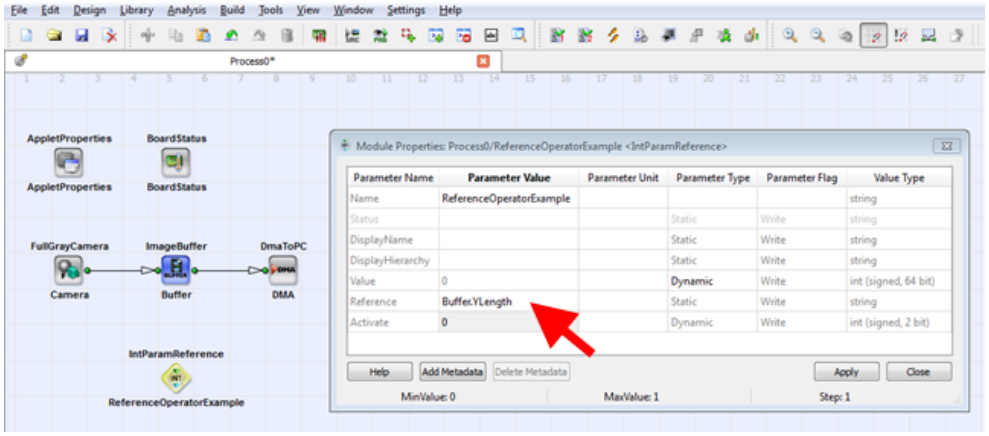
2. Double click on the reference operator instance to open its module properties.



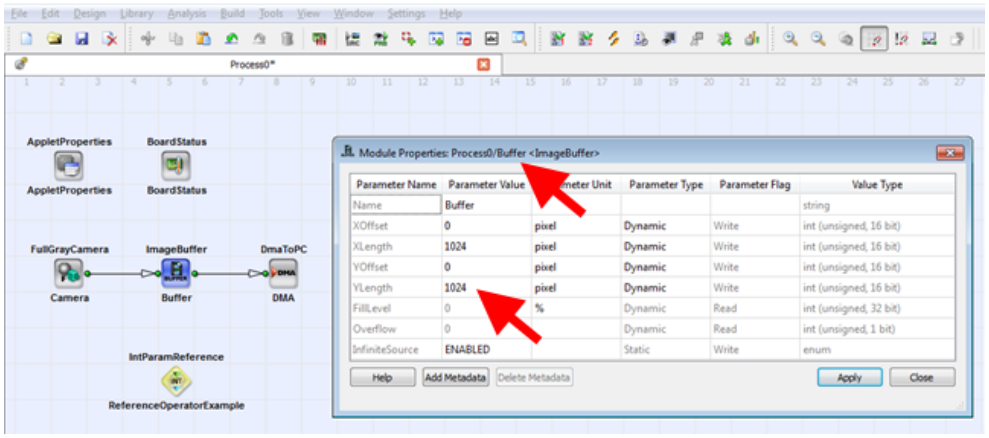

3. In Parameter *Reference*, specify the source parameter that you want the reference operator instance to connect to.

In the example below, an instance of the operator *IntParamReference* with the name *Reference Operator Example* connects to the parameter *YLength* in module *Buffer*. In this example, both modules (*Reference Operator Example* and *Buffer*) are located on the same hierarchical level.

Path to referenced parameter:




Referenced parameter:




**Syntax for Referencing**

The syntax is as follows:

```
{<Path>/}<Module>.<ParamName>
```

<Path> is the relative path to the hierarchical level where the referenced module "Module" is located. Use a slash as separator between different hierarchical levels. Define the path as relative path **from the hierarchical level the reference operator instance is located in**.

<Module> is the name of the module (operator instance or hierarchical box) that contains the referenced parameter.

<ParamName> is the actual name of the referenced parameter.

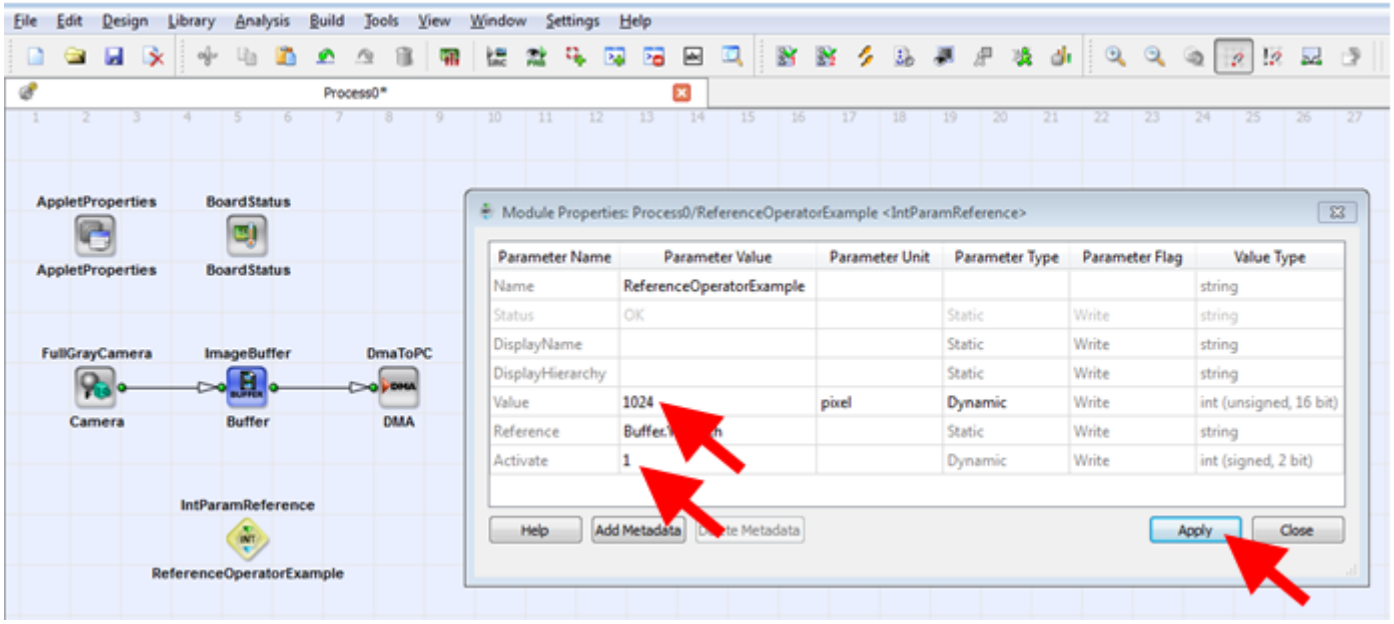
If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/SetParam.Value`).

You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules.

4. Set Parameter *Activate* to "Yes".

5. Click **Apply**. The value of the referenced parameter is displayed in parameter *Value* (wich is the map parameter).

In our example, the value of parameter *YLength* in operator *Buffer* is now also available in parameter *Value* of the reference operator instance:




With parameter *Value* in your reference operator, you can do many things.

You can, for example, make the value you have in parameter *Value* available in yet another module (operator instance or hierarchical box) anyplace in your design. To do so:

6. Define the name of a completely new parameter that is to bear the same value as parameter *Value*. You do this in parameter *DisplayName* of the reference operator. You are completely free in defining the name.

7. Define the module within your design that is to bear this new parameter. You do this in parameter *DisplayHierarchy* of the reference operator.

### Syntax for Setting up Path to Display Target

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

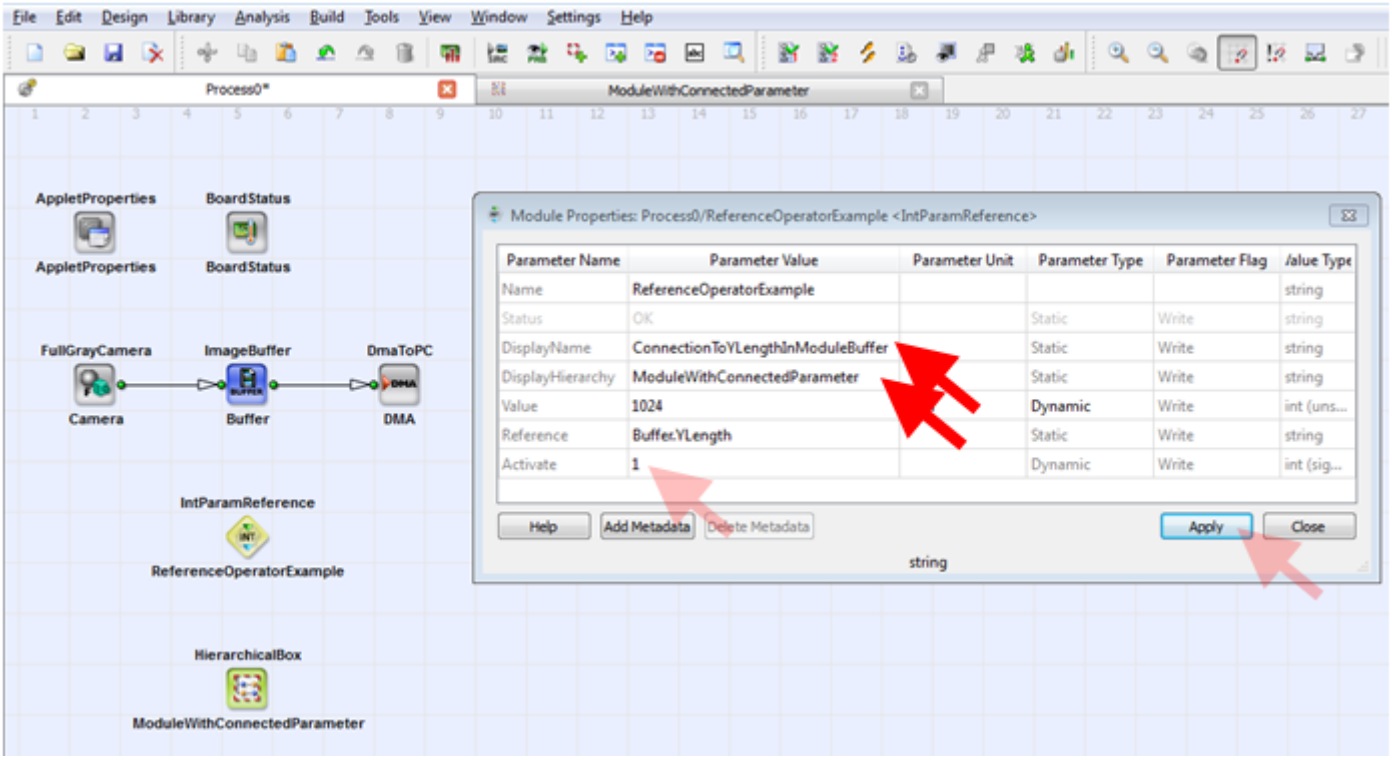
```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

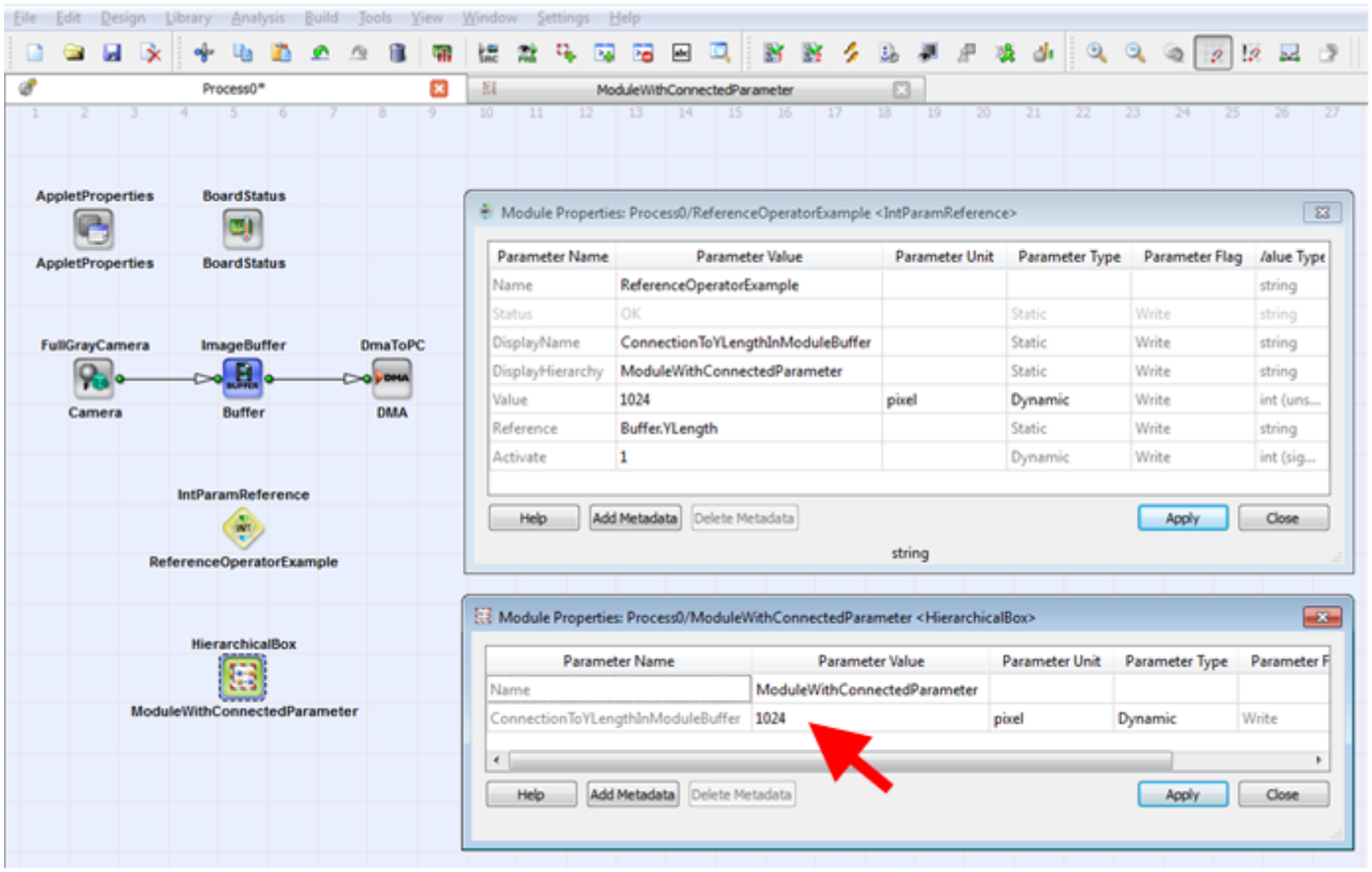
<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

In our example, parameter *Value* of the reference operator instance is connected to a new parameter *ConnectionToYLengthInModuleBuffer* of module *ModuleWithConnectedParameter* :



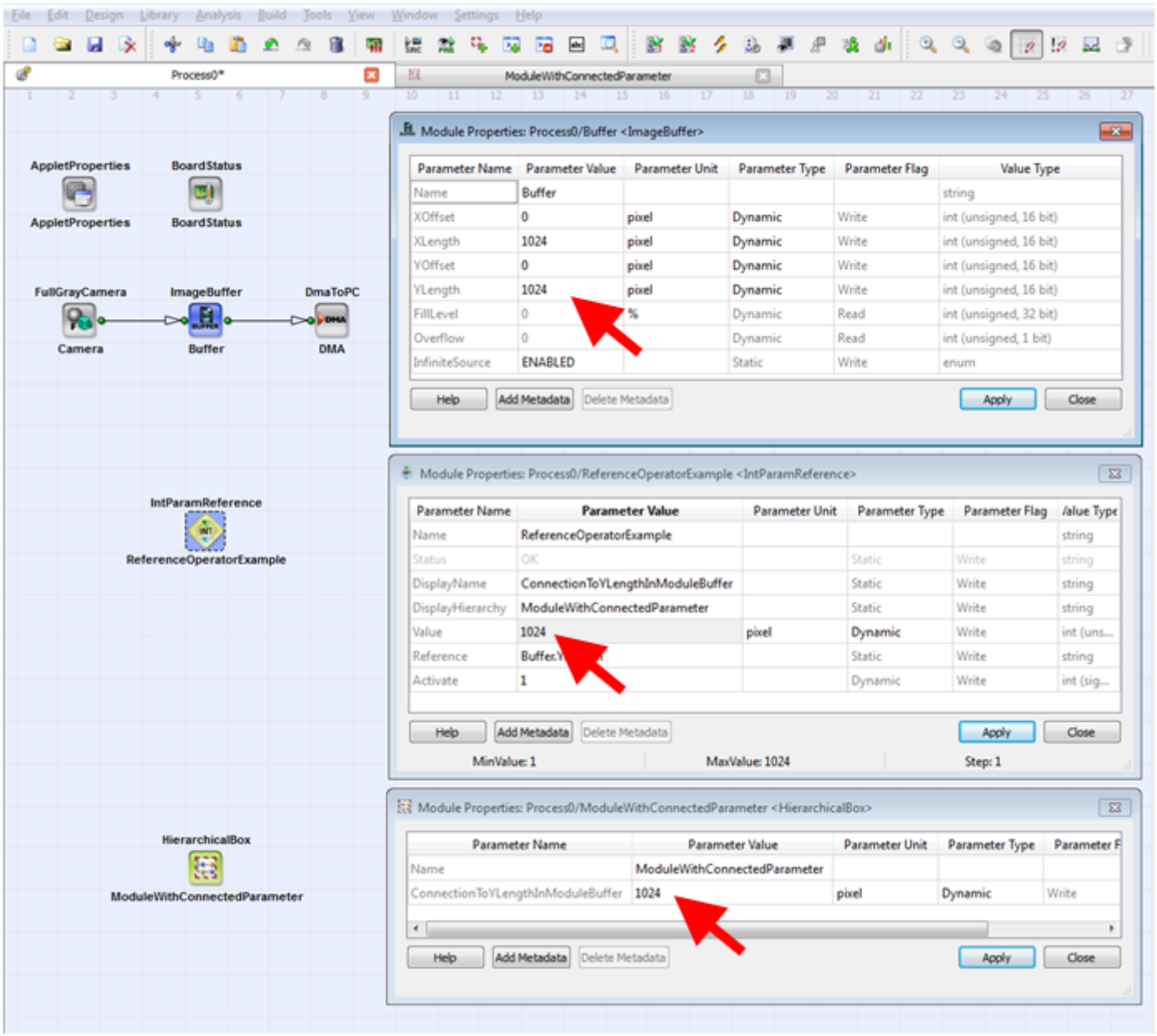

If parameter *Activate* is set to "Yes", after clicking the **Apply** button, the new parameter is displayed with the correct value in the connected module:




Thus, the new parameter *ConnectionToYLengthInModuleBuffer* of module *ModuleWithConnectedParameter* has the same value as parameter *YLength* of module *Buffer*.

The mapping is done by the reference operator instance.

If the connected parameters of any of the three modules is reset, the value is reset in all other modules, too:




**Hierarchical Boxes:**

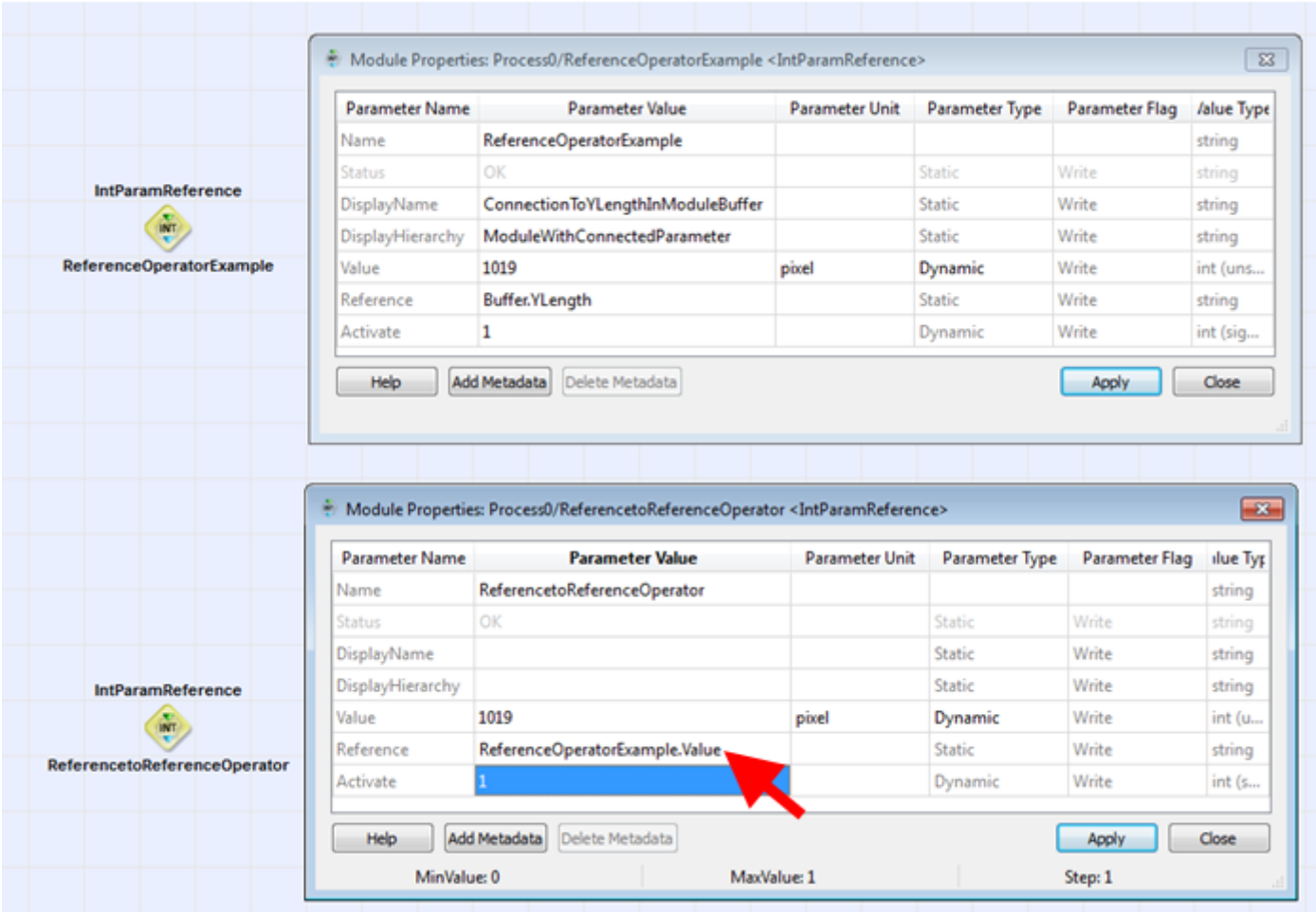
The *Reference* operators of the *Parameters* library allow you to make a parameter of your design available on any hierarchical level (of the design) you want. This is especially helpful if you want to allow parametrization of parameters nestled deeply within protected hierarchical boxes during runtime (i.e., if you want to make them available as dynamic write parameters). To allow easy parametrization of hierarchical boxes, hierarchical boxes can carry parameters now (as parameters in the *Properties* of the hierarchical box itself).

The *Reference* operators allow to set paths from the surface of a hierarchical box (*Properties* of the hierarchical box) to specific parameters which may be deeply embedded. This way, even parameters in protected hierarchical boxes (or in protected hierarchical boxes that are nestled in other protected hierarchical boxes) can be set and reset during runtime, since their parametrization interface is lifted up in the hierarchy to a hierarchical level that is not access restricted.

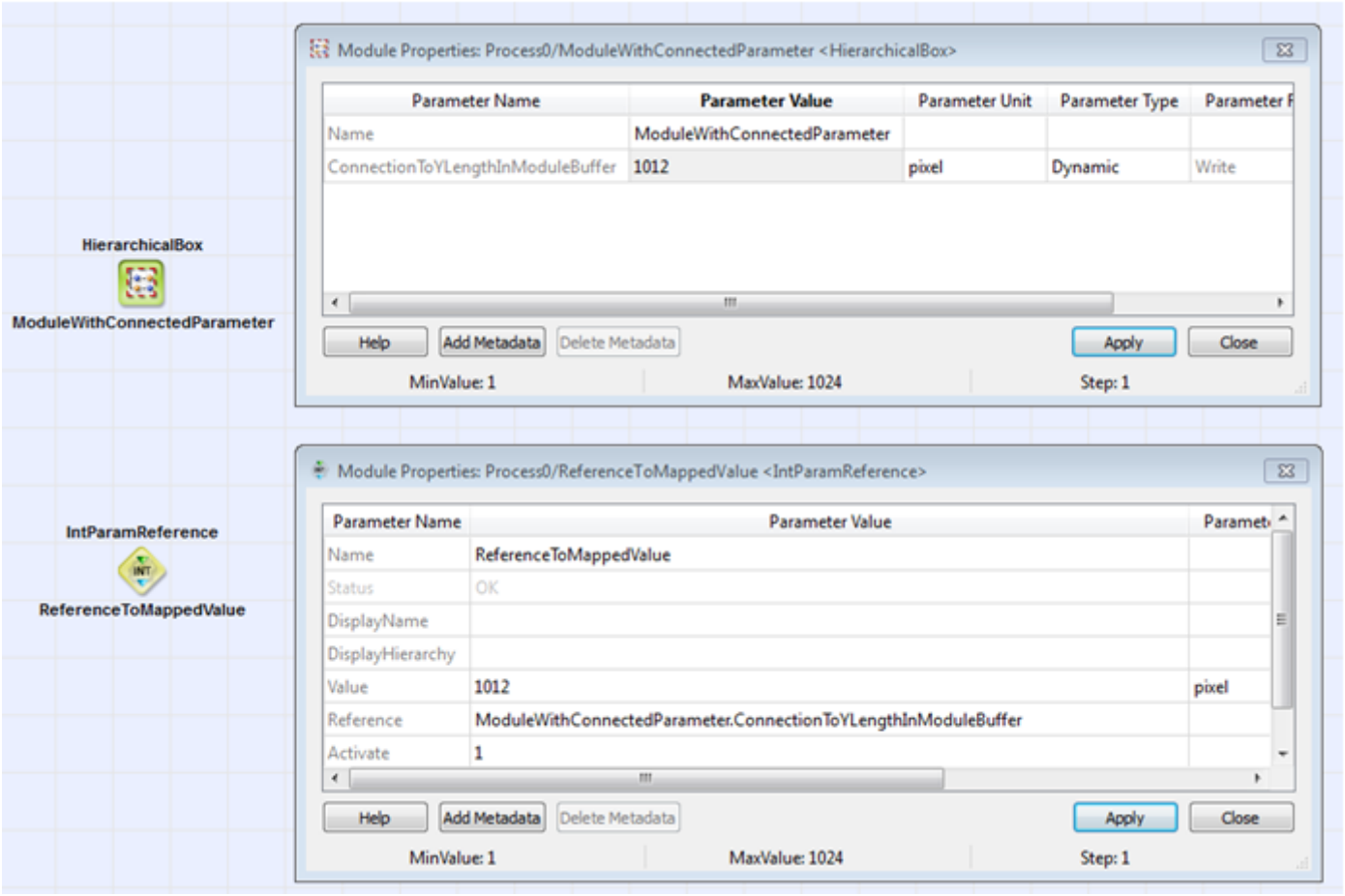
Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* (or *Field*) in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy* of the reference operator. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.

In our example, this means you can reference parameter *Value* in module *ReferenceOperatorExample* as well as parameter *ConnectionToYLengthInModuleBuffer* in module *ModuleWithConnectedParameter*.

Reference to parameter *Value* of reference operator instance *ReferenceOperatorExample*:




Reference to parameter *ConnectionToYLengthInModuleBuffer* of module *ModuleWithConnectedParameter*:






**Don't Create Reference Loops**

Make sure you don't implement reference loops, like *ModuleA.Value* referencing *ModuleB.Value*, and *ModuleB.Value* referencing *ModuleA.Value*.

Via the six available reference operators, all data types are provided that can occur as parameter values in protected hierarchical boxes.

**********

**The Translation Operators**

The translation operators of the parameters library allow translated access to parameters of one or several other operators.

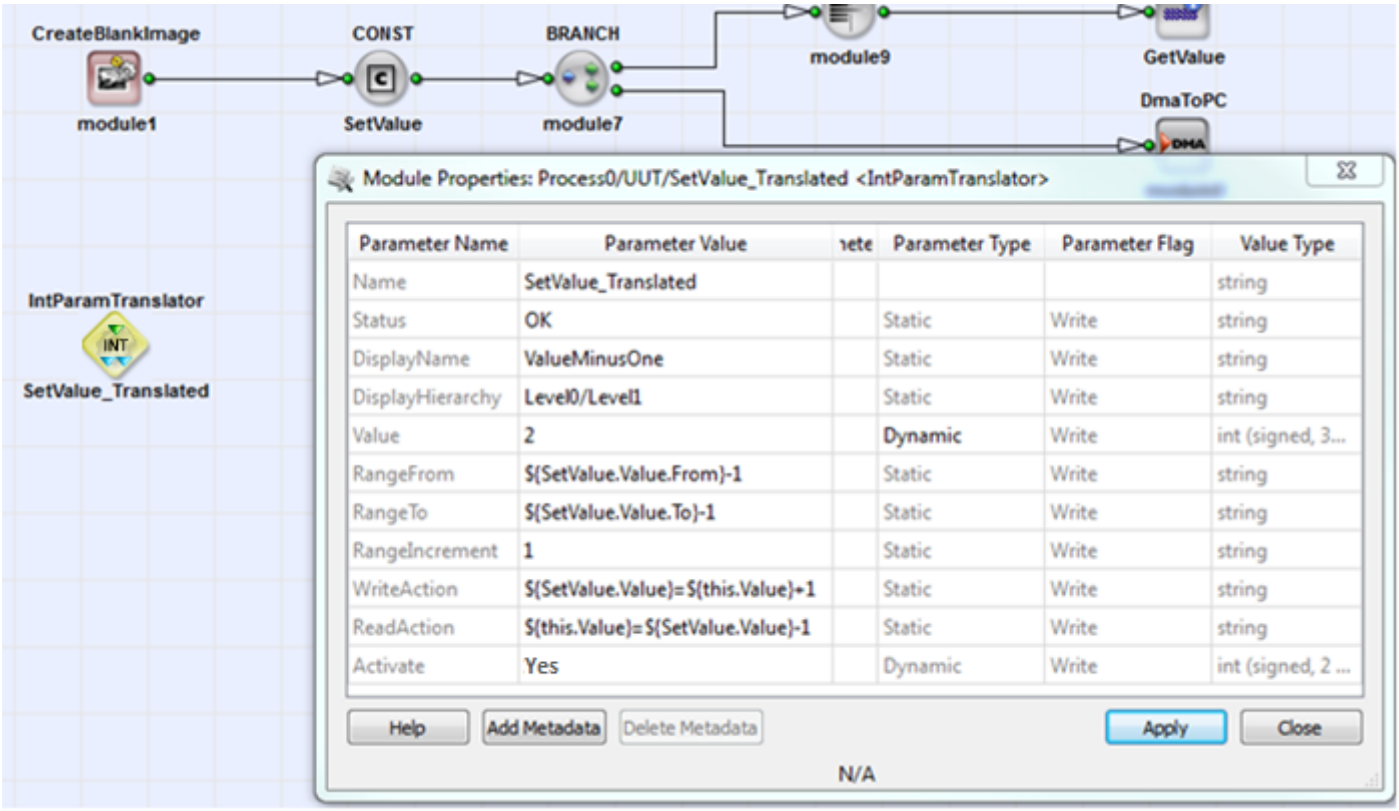
For calculation, you define formulas. The syntax for these formulas complies to GenICam API version 2.0 and is described below.

**Read Action:**

In parameter *ReadAction*, you define what happens for a read access to parameter *Value* of the translation operator.

One of the main differences to the reference operators is that translation operators don't simply hold a value. Instead, the value of parameter *Value* is calculated out of a formula. This formula you define in parameter *ReadAction*.

As soon as there happens a read access to parameter *Value*, the formula you defined in *ReadAction* is carried out and the result is forwarded to the element asking for the value.




The formula can contain values of operator parameters anywhere in the design. As soon as one of these values changes, the calculation of translation operator's parameter *Value* will end up with a new result (different to the value parameter *Value* held before).

The formula can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.

**Write Action**

In parameter *WriteAction*, you define what happens during a write access to parameter *Value* of the translation operator.

In contrast to the reference operators you do not simply forward the value of parameter *Value* to another parameter in the design. Instead, the value of parameter *Value* is used for a set of write actions you define in parameter *WriteAction* of the translation operator. Write actions are composed of one or several equations. With the left side of these equations you define which parameter of which operator receives the result of the calculation. If you define more than one equation separate the equations via semicolon.

As soon as the there happens a write access to parameter *Value*, the formula(s) on the right side of the equations you defined in parameter *WriteAction* is/are carried out and the result(s) is/are forwarded to the parameters specified on the left side of the equations.

The formula can refer to values of operator parameters anywhere in the design. As soon as one of these values changes, the calculation results will differ when you re-write the same value as before to the *Value* parameter.

The formula can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.

**Data Types:**

The operator you select defines the data type of its parameter *Value*, i.e.:

Operator *IntParamTranslator* -> integer value

Operator *FloatParamTranslator* -> float integer value

Operator *EnumParamTranslator* -> value of an enumeration

However, although the operator defines the type of its parameter *Value*, it does not restrict the data types of the parameters which are accessed during the write or read operations. For example, operator *IntParamTranslator* defines that its parameter *Value* is of type integer; but the formulas defined in the operator can contain access to, e.g., parameters of type double.

**Formulas for Calculation of Value Range**

You can also add formulas to calculate the current value range of parameter *Value*. You define the according formulas in the parameters *RangeFrom*, *RangeTo*, and *RangeIncrement* (step size).

The formulas can contain values of module parameters anywhere in the design, as well as specific properties of these parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.



### Syntax for Translation Operator Parameters

The syntax complies to the GenICam API standard in version 2.0. The allowed formula elements are identical to the formula elements defined in the GenICam standard. Find an explicite description in the individual documentation on operators *FloatParamTranslator* and *IntParamTranslator*.

**Paths to Parameters**

To access an operator parameter any place within your design, you need to provide the path to this parameter in your formula.

For access to an operator's parameter, you use the following construct:

```
${PathToModule/Module.ParamName}
```

"PathToModule": Here, you define the relative path to the operator who's parameter you want to access. The path is relative to the hierarchical level the translation operator itself is located. You also define the name of the accessed operator. Use a slash as hierarchy separator.

"Module": Name of module.

As name for the translation operator instance itself you use the name "this".



#### Keep your Modules Independent

It is not allowed to define a path towards a hierarchical level higher than the hierarchical level the translation operator is located. This rule follows the logic that a hierarchical module is not allowed to know anything about the environment it is instantiated in, because only in this case it can be used as a freely relocatable and replicable module.

**Access to Parameter Properties**

The formulas can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, or step size:

- `${PathToModule/Module.ParamName.From}` or `${PathToModule/Module.ParamName.Min}`: minimal valid value of parameter `PathToModule/Module.ParamName`.

- `${PathToModule/Module.ParamName.To}` or `${PathToModule/Module.ParamName.Max}`: maximum valid value of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Inc}`: Increment (step size) between two valid values of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Enum("EnumName")}`: Integer value of enumeration name `EnumName`.

**Syntax for Write Access Formulas**

Equations for write actions you define in parameter WriteAction. They have the following syntax:

```
${PathToTargetModule/TargetModule.TargetParamName}=GenICamFormula
```

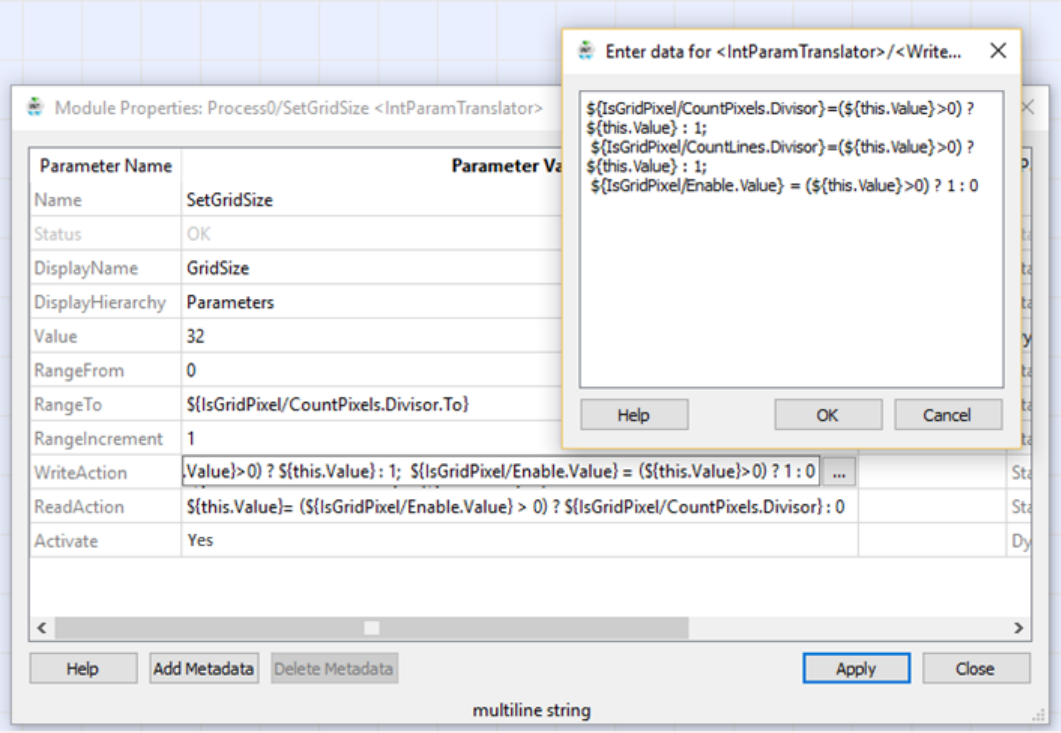
Here, you can define multiple equations for multiple target parameters. Use a semicolon as separator between the individual equations.

"this" refers to the translation operator instance itself.

Example for a WriteAction Equation:

```
${SetValue.Value}=${this.Value}+1
```

Example:




**Syntax for Read Access Equations**

Equations for read actions you define in parameter ReadAction. They have the following syntax:

```
${this.Value}=GenICamFormula
```

Example for a ReadAction equation:

```
${this.Value}=${SetValue.Value}-1
```

The following list summarizes all Operators of Library Parameters

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | EnumParamReference | Generates map parameter out of a reference parameter that is of data type VA_ENUM. | Version 3.0.1 |
|  | EnumParamTranslator | Generates translated map parameter of type VA_ENUM. Uses parameter of type VA_ENUM to create multiple translated map parameters in other modules. | Version 3.0.1 |
|  | EnumVariable | Generates a software variable. | Version 3.1 |
|  | FloatFieldParamReference | Generates a map parameter out of a reference parameter of type VA_DOUBLEFIELD. | Version 3.0.1 |
|  | FloatParamReference | Generates a map parameter out of a reference parameter of type VA_DOUBLE. | Version 3.0.1 |
|  | FloatParamTranslator | Generates translated map parameter of type VA_DOUBLE. Uses parameter of type VA_DOUBLE to create multiple translated map parameters in other modules. | Version 3.0.1 |
|  | FloatVariable | Generates a software variable. | Version 3.1 |
|  | IntFieldParamReference | Generates a map parameter out of a reference parameter of type VA_SINTFIELD or VA_UINTFIELD. | Version 3.0.1 |
|  | IntParamReference | Generates an Int64 map parameter out of a reference parameter of type VA_SINT, VA_UINT, or VA_ENUM. | Version 3.0.1 |
|  | IntParamTranslator | Generates translated map parameter of type VA_SINT. Uses parameter of type VA_SINT to create multiple translated map parameters in other modules. | Version 3.0.1 |
|  | IntVariable | Generates a software variable. | Version 3.1 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | IntFieldVariable | This operator generates a software variable field. | Version 3.5 |
|  | LinkProperties | Allows access to properties of connected link. | Version 3.0.1 |
|  | LinkParamTranslator | This operator allows reading or writing properties of a connected link and can perform write actions. | Version 3.3.0 |
|  | StringParamReference | Generates a map parameter (string) out of a reference parameter of type VA_STRING, VA_FILENAME, or VA_METADATA. | Version 3.0.1 |
|  | ResourceReference | This operator generates a map parameter (int) out of a reference to a device resource mapping. | Version 3.3.0 |
|  | IntParamSelector | This operator generates an Int64-map parameter which can be switched between several referenced parameters of type VA_SINT, VA_UINT, or VA_ENUM. | Version 3.3.0 |
|  | FloatParamSelector | This operator generates a map parameter which can be switched between several referenced parameters of type VA_DOUBLE. | Version 3.3.0 |

Table 28.2. Operators of Library Parameters

# 28.1. Operator EnumParamReference

**Operator Library: Parameters**

This operator generates a 1:1 map parameter (in parameter *Value*) out of a module parameter located anywhere in the design. You specify the referenced parameter (path and name) in parameter *Reference*.



**Availability**

To use the *EnumParamReference* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced parameter is of type VA_ENUM. The map parameter mirrors all properties of the referenced parameter and is also of type VA_ENUM.

You can define a target within the design hierarchy where the referenced parameter will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the reference operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the reference operator itself is located in).

This allows you to make a parameter of your design available on any hierarchical level (of the design) you want. For example, you can make the parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



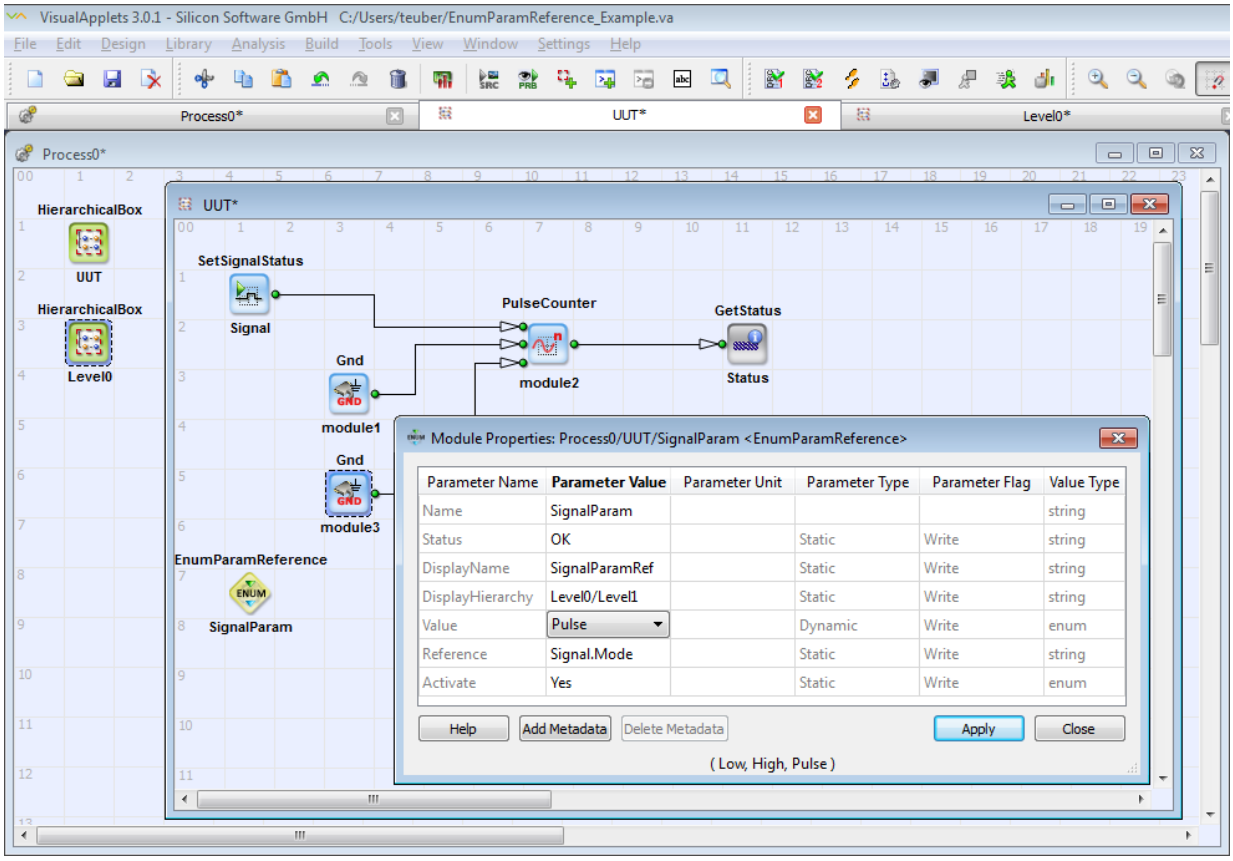
**All connected parameters update at once**

If one of the connected parameters is changed (referenced parameter, map parameter *Value*, or target parameter), the value is changed in all other connected parameters, too.

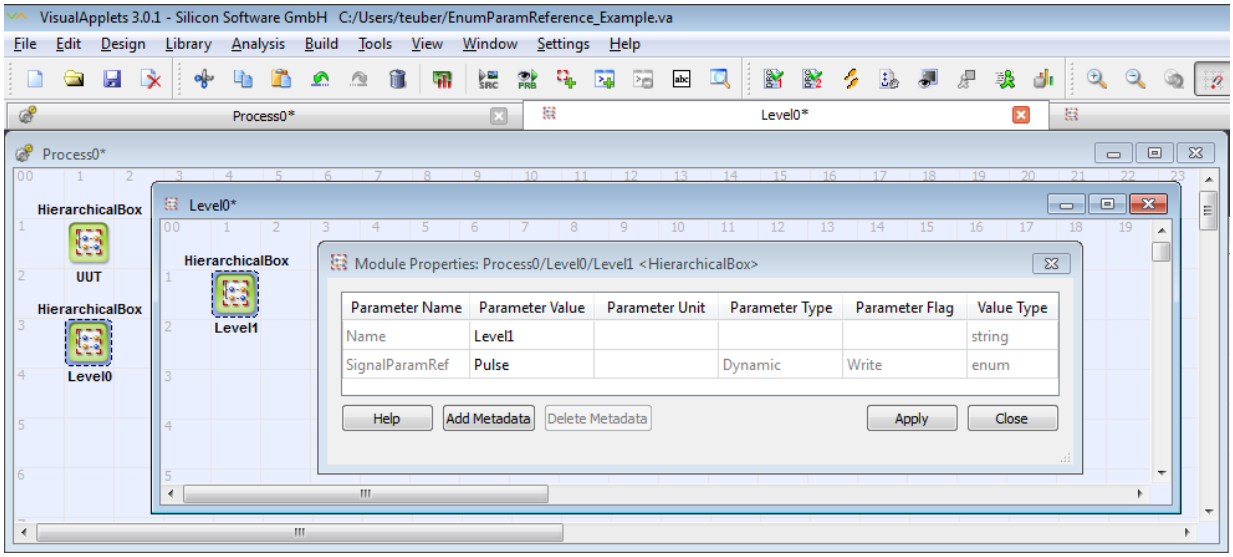
The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

Example:




Map parameter "SignalParamRef" is now available in Process0/Level0/Level1:




## 28.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

### 28.1.2. Supported Link Format

None

### 28.1.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |

Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.

This parameter is not part of the final applet.

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

If you want to grant access from and to the referenced parameter (defined in parameter *Reference*) at another point in the design (in addition to the reference operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.

If this field is empty, no new parameter is created.

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus,

| DisplayHierarchy |
|---|
| parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.<br><br><HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.<br><br>Example: `Level0/Level1` |

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit), unsigned integer (64 bit) |
| Map parameter for access from and to the referenced parameter. This parameter mirrors all properties of the referenced parameter. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the parameter *Value*. | |

| Reference | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string defining the referenced parameter: relative path to module, module name, and parameter name |

Here, you define the position of the referenced parameter within the design.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.



**Syntax for Referencing**

The syntax is as follows:

```
{<Path>/}<Module>.<ParamName>
```

<Path> is the relative path to the hierarchical level where the referenced module "Module" is located. Use a slash as separator between different hierarchical levels. Define the path as relative path **from the hierarchical level the reference operator instance is located in**.

<Module> is the name of the module (operator instance or hierarchical box) that contains the referenced parameter.

<ParamName> is the actual name of the referenced parameter.

If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/Signal.Mode`).

| Reference | |
|---|---|
| | You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules. |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameter (via map parameter *Value*) is activated.<br><br>*No* = Access to and from referenced parameter is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

### 28.1.4. Examples of Use

The use of operator EnumParamReference is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 14.1, 'Parameter Redirection'

  Examples - Demonstration how to use the parameter reference operators.

# 28.2. Operator EnumParamTranslator

**Operator Library: Parameters**

Operator *EnumParamTranslator* allows translated access (read and write) to parameters of one or several other operators. For calculation, you define formulas. The syntax for these formulas complies to GenICam API version 2.0.

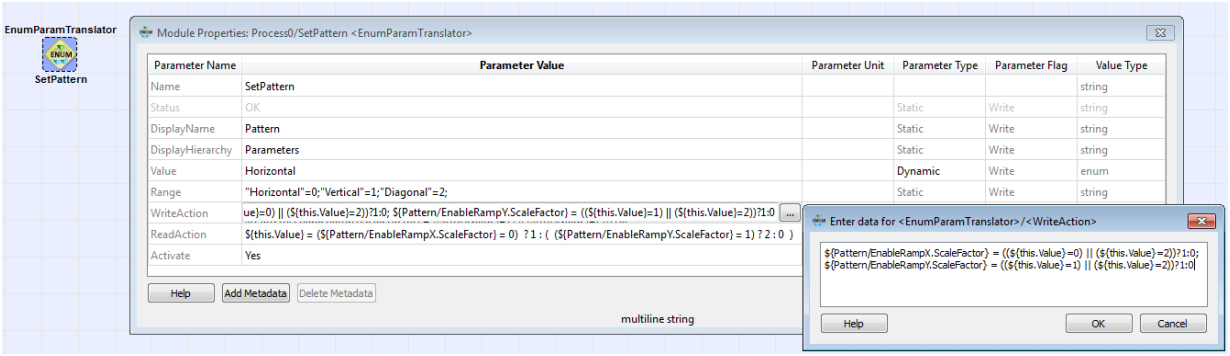


**Availability**

To use the *EnumParamTranslator* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

Operator *EnumParamTranslator* generates a map parameter of type VA_ENUM. The name of the map parameter is *Value*.

Although the operator defines the type of its parameter *Value* to be of type VA_ENUM, it does not restrict the data types of the parameters which are accessed during the write or read operations. For example, the formulas defined in the operator can contain access to parameters of type float.




**Read Action**

In parameter *ReadAction*, you define what happens for a read access to parameter *Value* of the translation operator. Parameter *Value* doesn't simply hold a value. Instead, the value of parameter *Value* is calculated out of a formula. This formula you define in parameter *ReadAction*. As soon as there happens a read access to parameter *Value*, the formula you defined in *ReadAction* is carried out and the result is forwarded to the element asking for the value.

The formula can contain values of operator parameters anywhere in the design. As soon as one of these values changes, the calculation of translation operator's parameter *Value* will end up with a new result (different to the value parameter *Value* held before). The formula can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.

**Write Action**

In parameter *WriteAction*, you define what happens during a write access to parameter *Value* of the translation operator. In contrast to the reference operators you do not simply forward the value of parameter *Value* to another parameter in the design. Instead, the value of parameter *Value* is used for a set of write actions you define in parameter *WriteAction* of the translation operator. Write actions are composed of one or several equations. With the left side of these equations you define which parameter of which operator receives the result of the calculation. If you define more than one equation, separate the equations via semicolon. As soon as the there happens a write access to parameter *Value*, the formula(s) on the right side of the equations you defined in parameter *WriteAction* is/are carried out and the result(s) is/are forwarded to the parameters specified on the left side of the equations.

The formula can refer to values of operator parameters anywhere in the design. As soon as one of these values changes, the calculation results will differ when you re-write the same value as before to

the *Value* parameter. The formula can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.



**Syntax for Translation Operator Parameters**

The syntax complies to the GenICam API standard in version 2.0. The allowed formula elements are identical to the formula elements defined in the GenICam standard.



**Paths to Parameters**

To access an operator parameter any place within your design, you need to provide the path to this parameter in your formula.

For access to an operator's parameter, you use the following construct:

```
${PathToModule/Module.ParamName}
```

"PathToModule": Here, you define the relative path to the operator who's parameter you want to access. The path is relative to the hierarchical level the translation operator itself is located. You also define the name of the accessed operator. Use a slash as hierarchy separator.

"Module": Name of module.

As name for the translation operator instance itself you use the name "this".



**Keep your Modules Independent**

It is not allowed to define a path towards a hierarchical level higher than the hierarchical level the translation operator is located. This rule follows the logic that a hierarchical module is not allowed to know anything about the environment it is instantiated in, because only in this case it can be used as a freely relocatable and replicable module.



**Access to Parameter Properties**

The formulas can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, or step size:

- `${PathToModule/Module.ParamName.From}` or `${PathToModule/Module.ParamName.Min}`: minimal valid value of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.To}` or `${PathToModule/Module.ParamName.Max}`: maximum valid value of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Inc}`: Increment (step size) between two valid values of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Enum("EnumName")}`: Integer value of enumeration name `EnumName`.



### Syntax for Write Access Equations

Equations for write actions you define in parameter WriteAction. They have the following syntax:

```
${PathToTargetModule/TargetModule.TargetParamName}=GenICamFormula
```

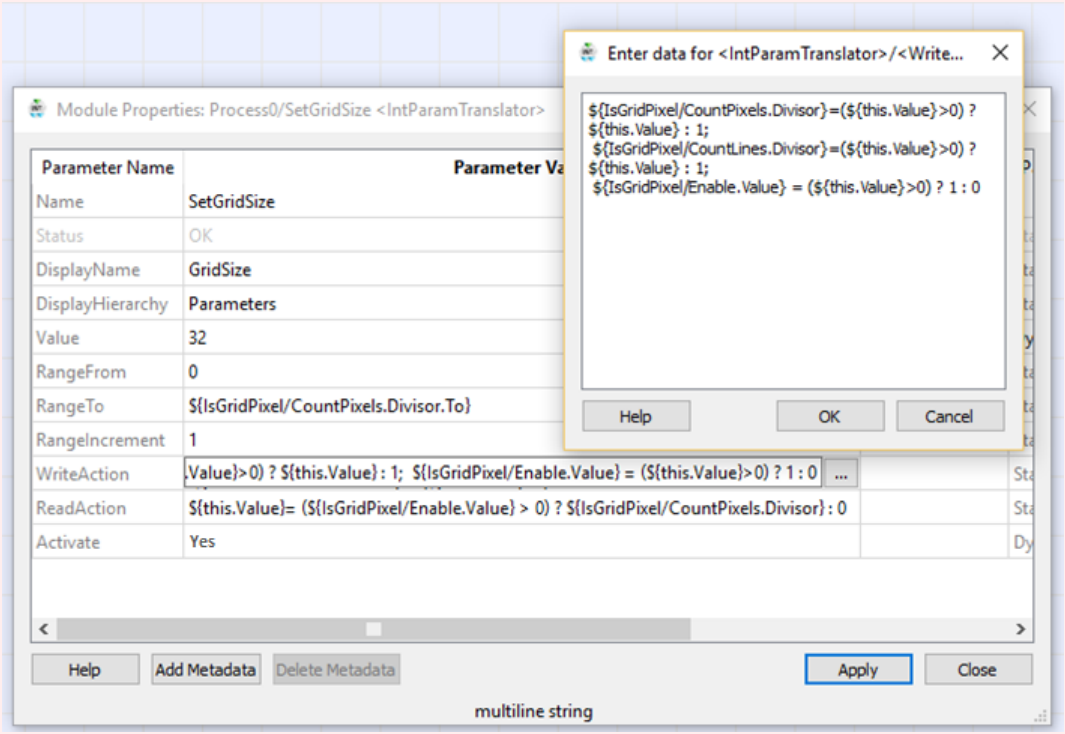
Here, you can define multiple equations for multiple target parameters. Use a semicolon as separator between the individual equations.

"this" refers to the translation operator instance itself.

Example for a WriteAction Equation:

```
${SetValue.Value}=${this.Value}+1
```

Example:






### Syntax for Read Access Equation

The equation for the read action you define in parameter *ReadAction*. The equation always starts with "${this.Value}=" . It has the following syntax:

```
${this.Value}=GenICamFormula
```

Example for a ReadAction equation:

```
${this.Value}=${SetValue.Value}-1
```

**Syntax for Defining the Enumeration in Parameter "Range"**

Use the following Syntax: "<String1>"=0;"<String2>"=1; "<String3>"=2;

Example:

| Range | "Horizontal"=0;"Vertical"=1;"Diagonal"=2; |
|---|---|

**Referencing Map Parameters**

You can define a target within the design hierarchy where map parameter *Value* will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the translation operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the translation operator itself is located in).

This allows you to make map parameter *Value* accessible on any hierarchical level (of the design) you want. This is especially helpful when workin with protected hierarchical boxes. You can make the parameter available directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a translation operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

### 28.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

### 28.2.2. Supported Link Format

None

### 28.2.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |
| Displays the error status. If parameter *Activate* is set to *Yes*, the other operator settings are checked. This parameter displays the result of this check, i.e., either *OK* or an error message. | |

| Status |
|---|
| This parameter is not part of the final applet. |

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| If you want to grant access from and to the parameter *Value* at another point in the design, you need to define a new parameter. Specify here the name of this new parameter.<br><br>If this field is empty, no new parameter is created. | |

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | Default |
| **Range** | |
| Parameter for access to other parameters via the equations specified for read and write accesses. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the parameter *Value*. | |

| Range | |
|---|---|
| **Type** | static write parameter |
| **Default** | "Default"=0; |
| **Range** | |
| Definition of name-value combinations. | |

| WriteAction | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | |
| Definition of equations for translation during write access to parameter *Value*.<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting.<br>To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and type "${". More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'. | |

| ReadAction | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | |
| Definition of equation for translation during read access to parameter *Value*.<br><br>The equation always starts with "${this.Value}=" .<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting.<br>To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and type "${". More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'. | |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameters (via read and write equations) is activated.<br><br>*No* = Access to and from referenced parameters (via read and write equations) is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.2.4. Examples of Use

The use of operator EnumParamTranslator is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 14.2, 'Parameter Translation'

  Examples - Demonstration how to use the parameter translation operators for manipulation of parameters.

# 28.3. Operator EnumVariable

**Operator Library: Parameters**

Operator *EnumVariable* generates a software variable.



**Availability**

To use the *EnumVariable* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

You can define a target within the design hierarchy where the variable will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the operator itself is located in).

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

## 28.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

## 28.3.2. Supported Link Format

None

## 28.3.3. Parameters

| **Status** | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |
| Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.<br><br>This parameter is not part of the final applet. | |

| **DisplayName** | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| If you want to grant access from and to the parameter at another point in the design (in addition to the operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.<br><br>If this field is empty, no new parameter is created. | |

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the operator instance. (Keep in mind this is not possible if the operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | Default |
| **Range** | |

Map parameter for access from and to the variable. This parameter mirrors all properties of the variable.

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |

Enter here the description for the parameter *Value*.

| Range | |
|---|---|
| **Type** | static write parameter |
| **Default** | "Default"=0; |
| **Range** | |

| **Range** |
| --- |
| Definition of name-value combinations. |

| **Activate** | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from the variable (via map parameter *Value*) is activated.<br><br>*No* = Access to and from the variable is de-activated and parameter *Status* is disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.4. Operator FloatFieldParamReference

**Operator Library: Parameters**

This operator generates a 1:1 map parameter (in parameter *Field*) out of a field parameter located anywhere in the design. You specify the referenced parameter (path and name) in parameter *Reference*.



**Availability**

To use the *FloatFieldParamReference* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced field parameter can be of one of type VA_DOUBLEFIELD. The map parameter mirrors all properties of the referenced parameter.

Since there are no field parameters specified in the GenICam standard, the created GenICam API and the created Framegrabber API code (FgLib) vary on this point. In the GenICam API code, the two parameters <DisplayName>_Index und <DisplayName>_Value are created.

You can define a target within the design hierarchy where the referenced parameter will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the reference operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the reference operator itself is located in).

This allows you to make a field parameter of your design available on any hierarchical level (of the design) you want. For example, you can make the field parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Field* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



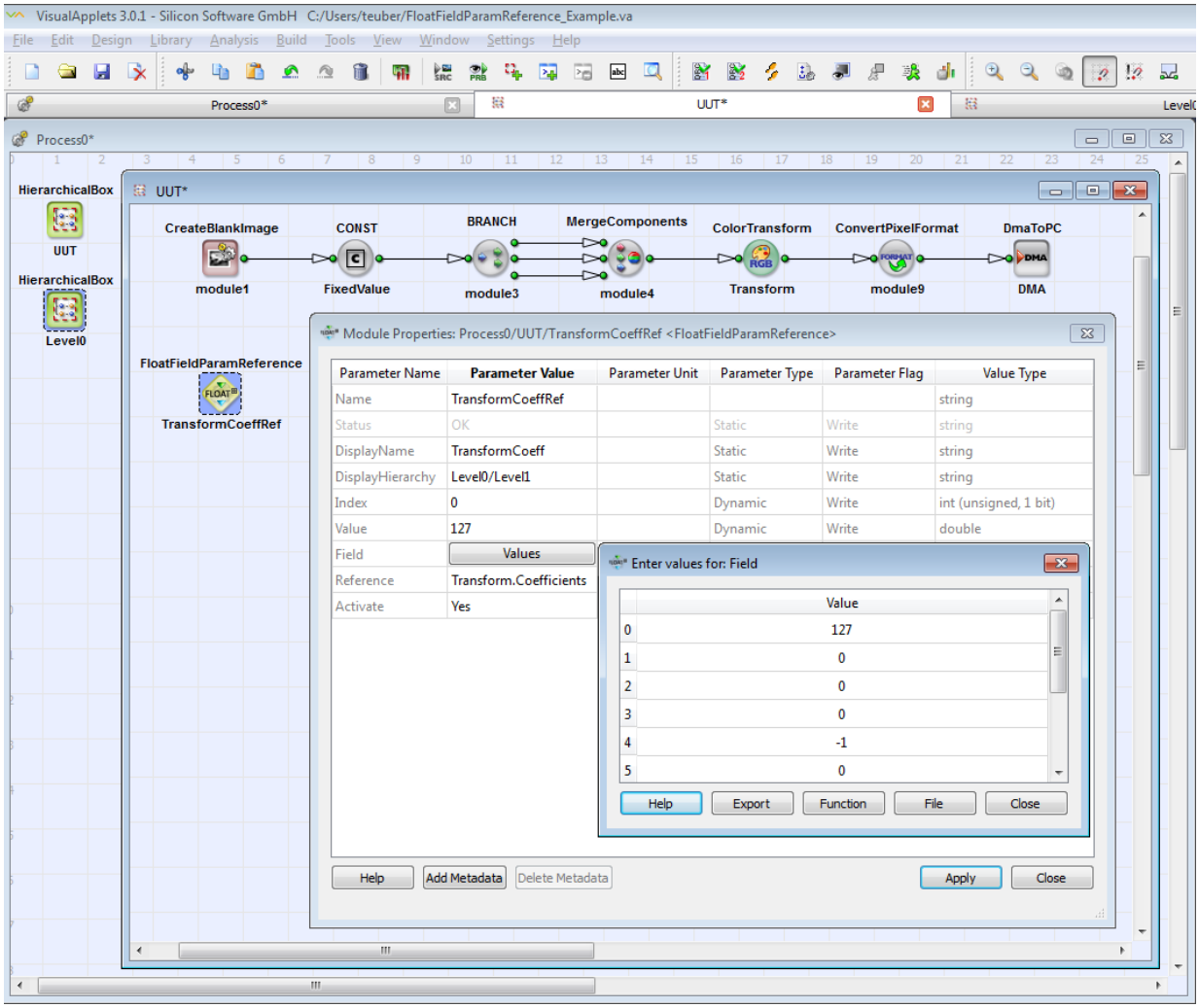
**All connected parameters update at once**

If one of the connected parameters is changed (referenced parameter, map parameter *Field*, or target parameter), the value is changed in all other connected parameters, too.

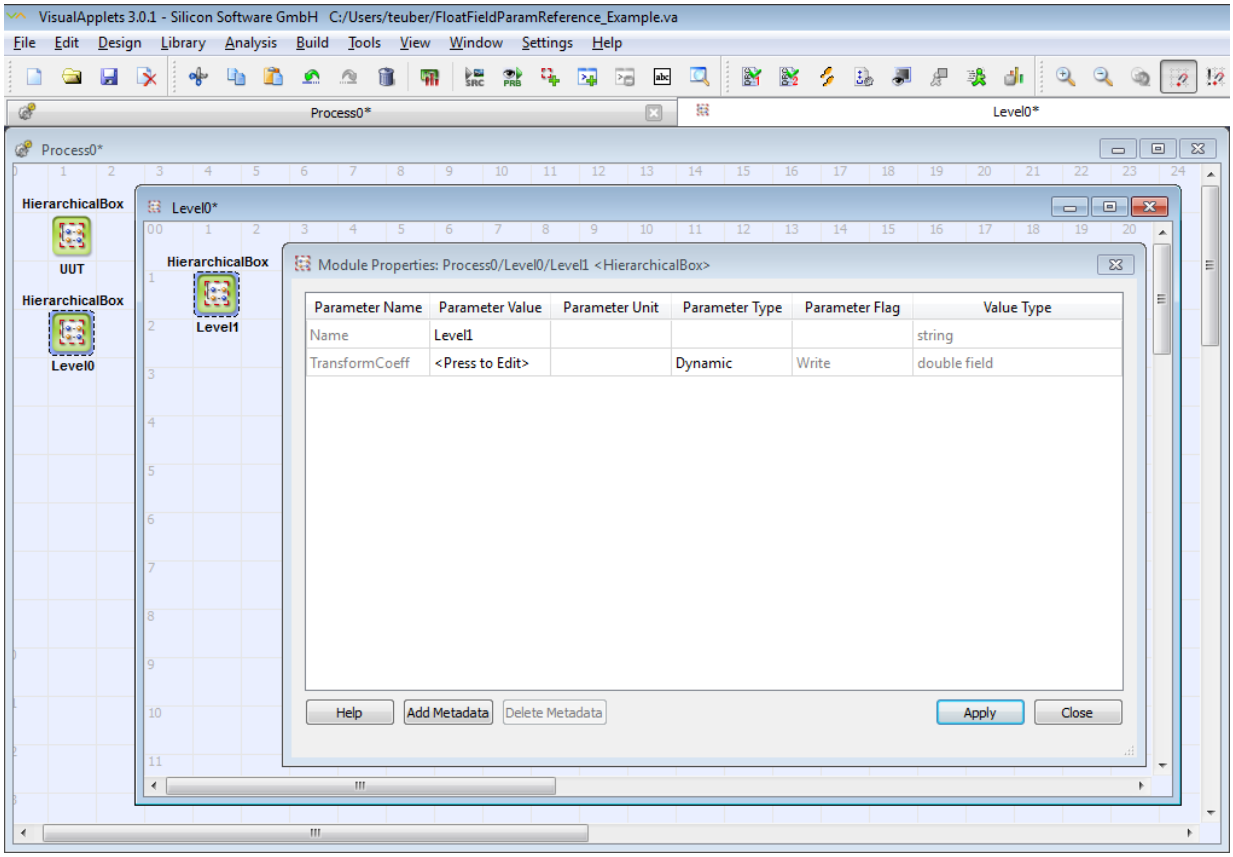
The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

Example:




The map parameter `TransformCoeff` is now available in Process0/Level0/Level1:




## 28.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

## 28.4.2. Supported Link Format

None

## 28.4.3. Parameters

| **Status** | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |
| Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.<br><br>This parameter is not part of the final applet. | |

| **DisplayName** | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

| DisplayName |
|---|
| If you want to grant access from and to the referenced field parameter (defined in parameter *Reference*) at another point in the design (in addition to the reference operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.<br><br>If this field is empty, no new parameter is created. |

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.)



### Syntax for Setting up Path to Display Target

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

| Index | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | UInt |

Set the field index here.

| Value | |
|---|---|
| **Type** | dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit), unsigned integer (64 bit) |

| Value |
|---|
| Set or read the field entry addressed by parameter *Index*. |

| Field | |
|---|---|
| **Type** | static,dynamic, write, read parameter |
| **Default** | none |
| **Range** | array |
| Map parameter for access to referenced field parameter. This parameter mirrors all properties of the referenced field parameter. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the parameter *Value*. | |

| Reference | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string defining the referenced parameter: relative path to module, module name, and parameter name |

Here, you define the position of the referenced parameter within the design.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.



**Syntax for Referencing**

The syntax is as follows:

```
{<Path>/}<Module>.<ParamName>
```

<Path> is the relative path to the hierarchical level where the referenced module "Module" is located. Use a slash as separator between different hierarchical levels. Define the path as relative path **from the hierarchical level the reference operator instance is located in**.

<Module> is the name of the module (operator instance or hierarchical box) that contains the referenced parameter.

<ParamName> is the actual name of the referenced parameter.

If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/Transform.Coefficients`).

You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules.

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |

| **Activate** | |
|---|---|
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameter (via map parameter *Value*) is activated.<br><br>*No* = Access to and from referenced parameter is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

### 28.4.4. Examples of Use

The use of operator FloatFieldParamReference is shown in the following examples:

- Section 14.1, 'Parameter Redirection'

  Examples - Demonstration how to use the parameter reference operators.

## 28.5. Operator FloatParamReference

**Operator Library: Parameters**

This operator generates a 1:1 map parameter (in parameter *Value*) out of a module parameter located anywhere in the design. You specify the referenced parameter (path and name) in parameter *Reference*.



**Availability**

To use the *FloatParamReference* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced parameter is of type VA_DOUBLE. The map parameter mirrors all properties of the referenced parameter and is also of type VA_DOUBLE.

You can define a target within the design hierarchy where the selected parameter will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the reference operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the reference operator itself is located in).

This allows you to make a parameter of your design available on any hierarchical level (of the design) you want. For example, you can make the parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



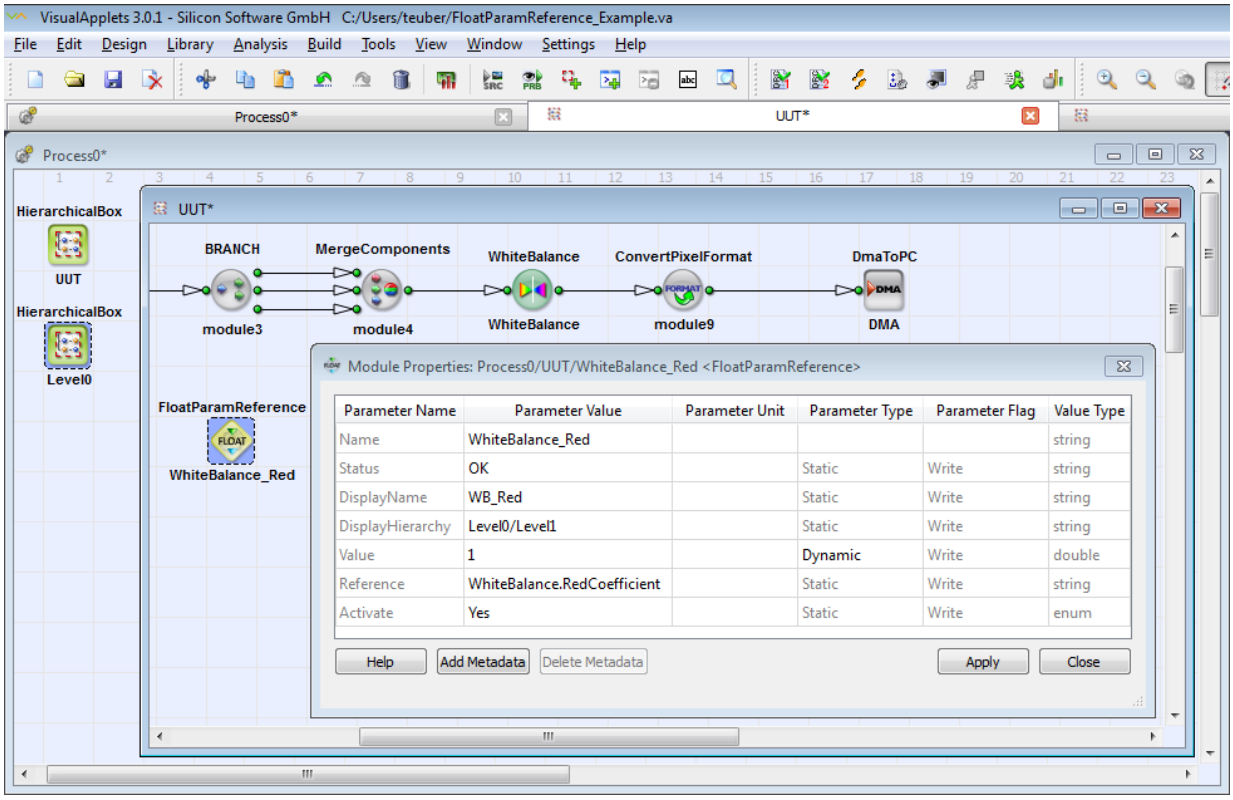
**All connected parameters update at once**

If one of the connected parameters is changed (referenced parameter, map parameter *Value*, or target parameter), the value is changed in all other connected parameters, too.

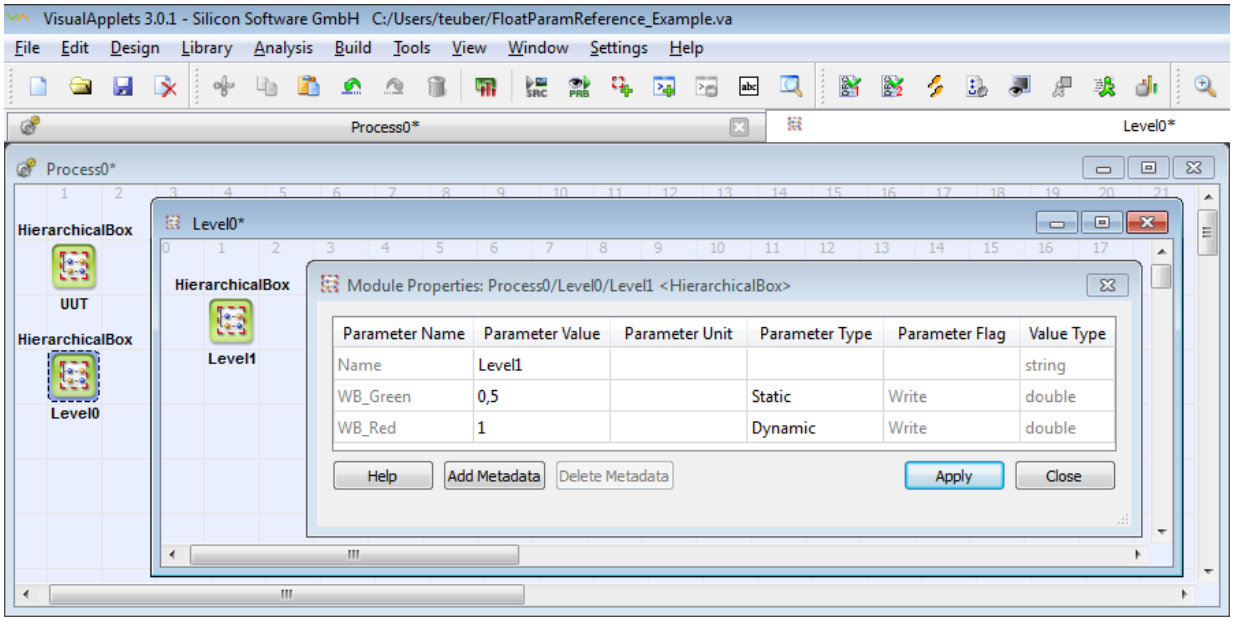
The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

Example:




Map parameter "WB_Red" is now available in Process0/Level0/Level1:




## 28.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

### 28.5.2. Supported Link Format

None

### 28.5.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |

Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.

This parameter is not part of the final applet.

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

If you want to grant access from and to the referenced parameter (defined in parameter *Reference*) at another point in the design (in addition to the reference operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.

If this field is empty, no new parameter is created.

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus,

| DisplayHierarchy |
|---|
| parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.<br><br><HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.<br><br>Example: `Level0/Level1` |

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | double |
| Map parameter for access from and to the referenced parameter. This parameter mirrors all properties of the referenced parameter. | |

| Unit | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |
| Enter here the unit for the parameter *Value*. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the parameter *Value*. | |

| Reference | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string defining the referenced parameter: relative path to module, module name, and parameter name |

Here, you define the position of the referenced parameter within the design.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.



**Syntax for Referencing**

The syntax is as follows:

```
{<Path>/}<Module>.<ParamName>
```

<Path> is the relative path to the hierarchical level where the referenced module "Module" is located. Use a slash as separator between different hierarchical levels. Define the path as relative path **from the hierarchical level the reference operator instance is located in**.

| Reference | |
|---|---|
| | <Module> is the name of the module (operator instance or hierarchical box) that contains the referenced parameter.<br><br><ParamName> is the actual name of the referenced parameter.<br><br>If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/WhiteBalance.RedCoefficient`).<br><br>You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules. |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameter (via map parameter *Value*) is activated.<br><br>*No* = Access to and from referenced parameter is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

### 28.5.4. Examples of Use

The use of operator FloatParamReference is shown in the following examples:

- Section 14.1, 'Parameter Redirection'

  Examples - Demonstration how to use the parameter reference operators.

# 28.6. Operator FloatParamTranslator

**Operator Library: Parameters**

Operator *FloatParamTranslator* allows translated access (read and write) to parameters of one or several other operators. For calculation, you define formulas. The syntax for these formulas complies to GenICam API version 2.0.

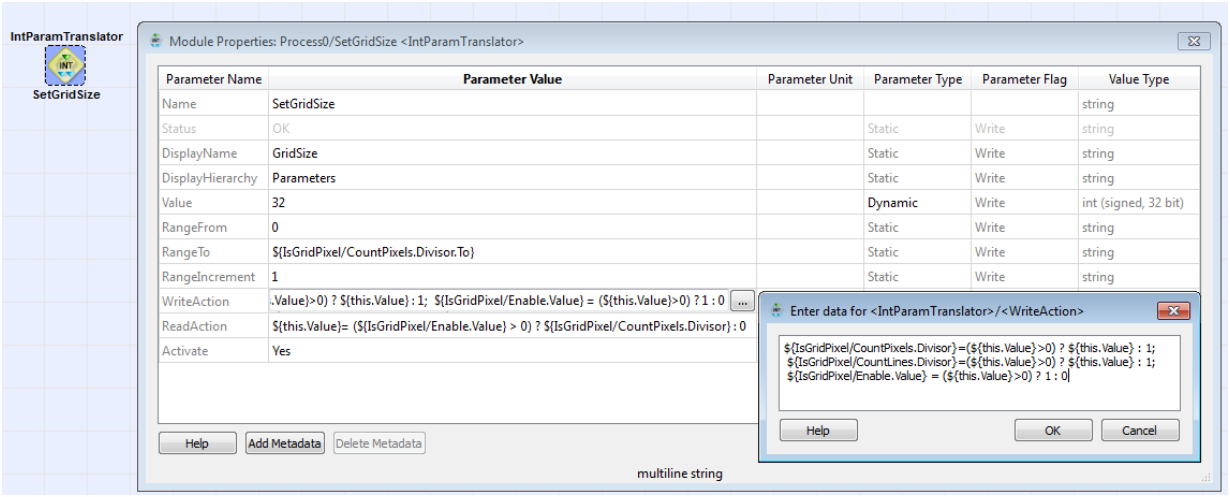


**Availability**

To use the *FloatParamTranslator* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

Operator *FloatParamTranslator* generates a map parameter of type VA_DOUBLE. The name of the map parameter is *Value*.

Although the operator defines the type of its parameter *Value* to be of type VA_DOUBLE, it does not restrict the data types of the parameters which are accessed during the write or read operations.




**Read Action**

In parameter *ReadAction*, you define what happens for a read access to parameter *Value* of the translation operator. Parameter *Value* doesn't simply hold a value. Instead, the value of parameter *Value* is calculated out of a formula. This formula you define in parameter *ReadAction*. As soon as there happens a read access to parameter *Value*, the formula you defined in *ReadAction* is carried out and the result is forwarded to the element asking for the value.

The formula can contain values of operator parameters anywhere in the design. As soon as one of these values changes, the calculation of translation operator's parameter *Value* will end up with a new result (different to the value parameter *Value* held before). The formula can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.

**Write Action**

In parameter *WriteAction*, you define what happens during a write access to parameter *Value* of the translation operator. In contrast to using the reference operators, you do not simply forward the value of parameter *Value* to another parameter in the design. Instead, the value of parameter *Value* is used for a set of write actions you define in parameter *WriteAction* of the translation operator. Write actions are composed of one or several equations. On the left-hand side of these equations you define which parameter of which operator receives the result of the calculation. If you define more than one equation, separate the equations via semicolon. As soon as the there happens a write access to parameter *Value*, the formula(s) on the right-hand side of the equations you defined in parameter

*WriteAction* is/are carried out and the result(s) is/are forwarded to the parameters specified on the left-hand side of the equations.

The formula can refer to values of operator parameters anywhere in the design. As soon as one of these values changes, the calculation results will differ when you re-write the same value as before to the *Value* parameter. The formula can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.

**Formulas for Calculation of Value Range**

You can also add formulas to calculate the current value range of parameter *Value*. You define the according formulas in the parameters *RangeFrom*, *RangeTo*, and *RangeIncrement* (step size). The formulas can contain values of module parameters anywhere in the design, as well as specific properties of these parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.



### Formula Syntax: Mathematical Operations

The syntax complies to the GenICam API standard in version 2.0. The allowed formula elements are identical to the formula elements defined in the GenICam standard:

| **Basic operations:** | |
|---|---|
| () | brackets |
| + - * / | addition, subtraction, multiplication, division |
| % | remainder |
| ** | power |
| & \| ^ ~ | bitwise: and / or / xor / not |
| <> = > < <= >= | logical relations: not equal / equal / greater / less / less or equal / greater or equal |
| && \|\| | logical and / logical or |
| << >> | shift left / shift right |

Table 28.3. Basic operations

**Conditional operator**

<condition> **?** <true_expr> **:** <false_expr>

Example:

${target.Value} = (${this.Value} > 0) ? 1 : 0;

| **Functions:** | |
|---|---|
| SGN(x) | return sign of x. Returns +1 for positive argument and -1 for negative argument |
| NEG(x) | swap sign of x |
| ABS(x) | return absolute value of x |
| SQRT(x) | return square root of x |
| TRUNC(x) | truncate x, which means returning the nearest integral value that is not larger in magnitude than x |

| | |
|---|---|
| FLOOR(x) | Round downward, returning the largest integral value that is not greater than x |
| CEIL(x) | Round upward, returning the smallest integral value that is not less than x |
| ROUND(x,precision) | round x to the number of decimal fractional digits given by precision, with halfway cases rounded away from zero |
| SIN(x) | return sine of an angle of x radians |
| COS(x) | return cosine of an angle of x radians |
| TAN(x) | return the tangent of an angle of x radians |
| ASIN(x) | return the principal value of the arc sine of x, expressed in radians |
| ACOS(x) | return the principal value of the arc cosine of x, expressed in radians |
| ATAN(x) | return the principal value of the arc tangent of x, expressed in radians |
| EXP(x) | return the base-e exponential function of x, which is e raised to the power x: ex |
| LN(x) | return the natural logarithm of x The natural logarithm is the base-e logarithm: the inverse of the natural exponential function (exp). |
| LG(x) | return the common (base-10) logarithm of x |
| E() | return Euler's number, 2.7182818284590451 |
| PI() | return circle constant, 3.1415926535897931 |

Table 28.4. Functions

Example:

${target.Value} = NEG(${this.Value})



### Paths to Parameters

To access an operator parameter any place within your design, you need to provide the path to this parameter in your formula.

For access to an operator's parameter, you use the following construct:

```
${PathToModule/Module.ParamName}
```

"PathToModule": Here, you define the relative path to the operator who's parameter you want to access. The path is relative to the hierarchical level the translation operator itself is located. You also define the name of the accessed operator. Use a slash as hierarchy separator.

"Module": Name of module.

As name for the translation operator instance itself you use the name "this".



**Keep your Modules Independent**

It is not allowed to define a path towards a hierarchical level higher than the hierarchical level the translation operator is located. This rule follows the logic that a hierarchical module is not allowed to know anything about the environment it is instantiated in, because only in this case it can be used as a freely relocatable and replicable module.



**Access to Parameter Properties**

The formulas can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, or step size:

- `${PathToModule/Module.ParamName.From}` or `${PathToModule/Module.ParamName.Min}`: minimal valid value of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.To}` or `${PathToModule/Module.ParamName.Max}`: maximum valid value of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Inc}`: Increment (step size) between two valid values of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Enum("EnumName")}`: Integer value of enumeration name `EnumName`.



**Syntax for Write Access Equations**

Equations for write actions you define in parameter WriteAction. They have the following syntax:

```
${PathToTargetModule/TargetModule.TargetParamName}=GenICamFormula
```

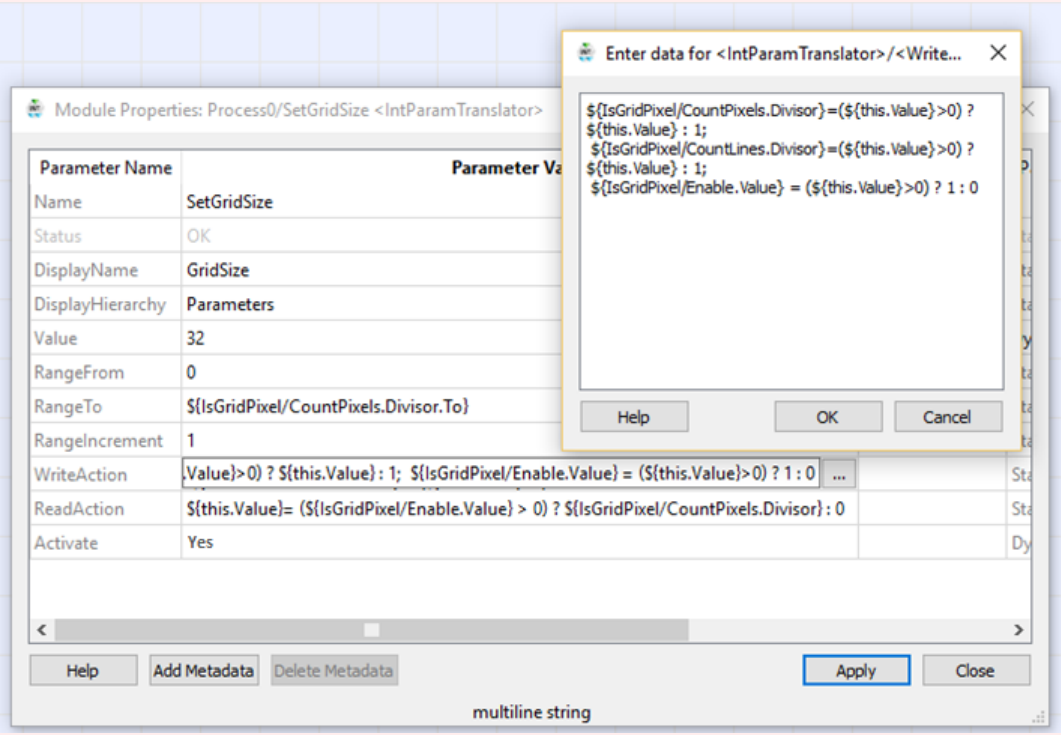
Here, you can define multiple equations for multiple target parameters. Use a semicolon as separator between the individual equations.

"this" refers to the translation operator instance itself.

Example for a WriteAction Equation:

```
${SetValue.Value}=${this.Value}+1
```

Example:






### Syntax for Read Access Equation

The equation for the read action you define in parameter *ReadAction*. The equation always starts with "${this.Value}=" . It has the following syntax:

```
${this.Value}=GenICamFormula
```

Example for a ReadAction equation:

```
${this.Value}=${SetValue.Value}-1
```

### Referencing Map Parameters

You can define a target within the design hierarchy where map parameter *Value* will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the translation operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the translation operator itself is located in).

This allows you to make map parameter *Value* accessible on any hierarchical level (of the design) you want. This is especially helpful when workin with protected hierarchical boxes. You can make the parameter available directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a translation operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

### 28.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

### 28.6.2. Supported Link Format

None

### 28.6.3. Parameters

| **Status** | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |
| Displays the error status. If parameter *Activate* is set to *Yes*, the other operator settings are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.<br><br>This parameter is not part of the final applet. | |

| **DisplayName** | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| If you want to grant access from and to the parameter *Value* at another point in the design, you need to define a new parameter. Specify here the name of this new parameter.<br><br>If this field is empty, no new parameter is created. | |

| **DisplayHierarchy** | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.<br><br>If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.) | |

| DisplayHierarchy |
|---|



### Syntax for Setting up Path to Display Target

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | double |
| Parameter for access to other parameters via the equations specified for read and write accesses. | |

| Unit | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |
| Enter here the unit for the parameter *Value*. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the parameter *Value*. | |

| RangeFrom | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0.1 |
| **Range** | |
| Definition of smallest valid value. You can also enter a formula here. | |

| RangeTo | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1.0 |
| **Range** | |
| Definition of biggest valid value. You can also enter a formula here. | |

| RangeIncrement | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0.1 |
| **Range** | |
| Definition of intervall between valid values (definition of stepsize). You can also enter a formula here. | |

| WriteAction | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | |
| Definition of equations for translation during write access to parameter *Value*.<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting.<br>To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and type "${". More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'. | |

| ReadAction | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | |
| Definition of equation for translation during read access to parameter *Value*.<br><br>The equation always starts with "${this.Value}=" .<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting.<br>To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and type "${". More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'. | |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameters (via read and write equations) is activated.<br><br>*No* = Access to and from referenced parameters (via read and write equations) is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.6.4. Examples of Use

The use of operator FloatParamTranslator is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 14.2, 'Parameter Translation'

  Examples - Demonstration how to use the parameter translation operators for manipulation of parameters.

# 28.7. Operator FloatVariable

**Operator Library: Parameters**

Operator *FloatVariable* generates a software variable.



**Availability**

To use the *FloatVariable* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

You can define a target within the design hierarchy where the variable will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the operator itself is located in).

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

## 28.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

## 28.7.2. Supported Link Format

None

## 28.7.3. Parameters

| **Status** | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |
| Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.<br><br>This parameter is not part of the final applet. | |

| **DisplayName** | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| If you want to grant access from and to the parameter at another point in the design (in addition to the operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.<br><br>If this field is empty, no new parameter is created. | |

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the operator instance. (Keep in mind this is not possible if the operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | double |

Map parameter for access from and to the variable. This parameter mirrors all properties of the variable.

| Unit | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |

Enter here the unit for the parameter *Value*.

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |

| Description |
|---|
| Enter here the description for the parameter *Value*. |

| RangeFrom | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0.1 |
| **Range** | |
| Definition of smallest valid value. You can also enter a formula here. | |

| RangeTo | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1.0 |
| **Range** | |
| Definition of biggest valid value. You can also enter a formula here. | |

| RangeIncrement | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0.1 |
| **Range** | |
| Definition of intervall between valid values (definition of stepsize). You can also enter a formula here. | |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from the variable (via map parameter *Value*) is activated.<br><br>*No* = Access to and from the variable is de-activated and parameter *Status* is disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.8. Operator IntFieldParamReference

**Operator Library: Parameters**

This operator generates a 1:1 map parameter (in parameter *Field*) out of a field parameter located anywhere in the design. You specify the referenced parameter (path and name) in parameter *Reference*.



**Availability**

To use the *IntFieldParamReference* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced field parameter can be of one of the following types: VA_SINTFIELD or VA_UINTFIELD. The map parameter mirrors all properties of the referenced parameter.

Since there are no field parameters specified in the GenICam standard, the created GenICam API and the created Framegtabber API code (FgLib) vary on this point. In the GenICam API code, the two parameters <DisplayName>_Index and <DisplayName>_Value are created.

You can define a target within the design hierarchy where the referenced parameter will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the reference operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the reference operator itself is located in).

This allows you to make a field parameter of your design available on any hierarchical level (of the design) you want. For example, you can make the field parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Field* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



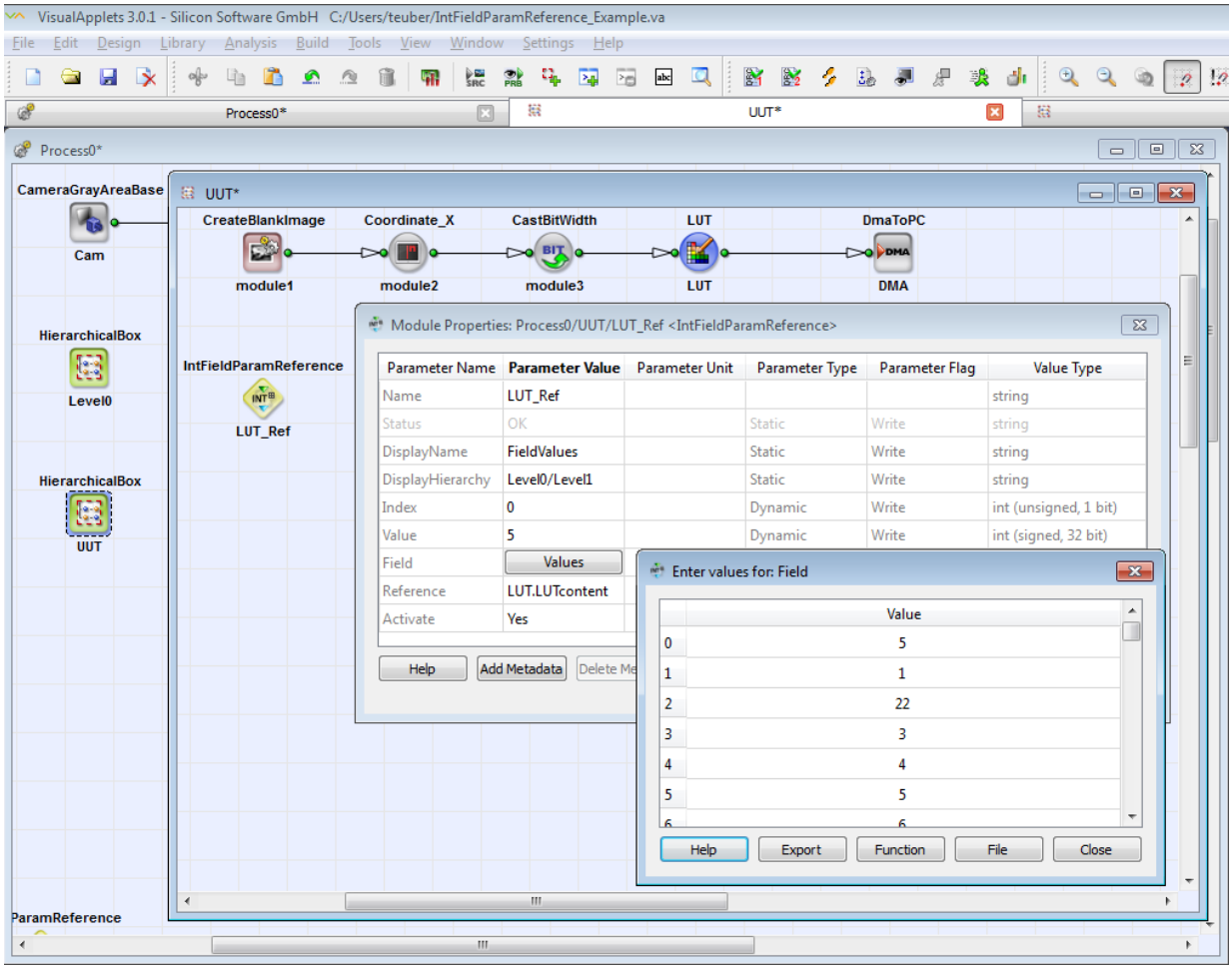
**All connected parameters update at once**

If one of the connected parameters is changed (referenced parameter, map parameter *Field*, or target parameter), the value is changed in all other connected parameters, too.

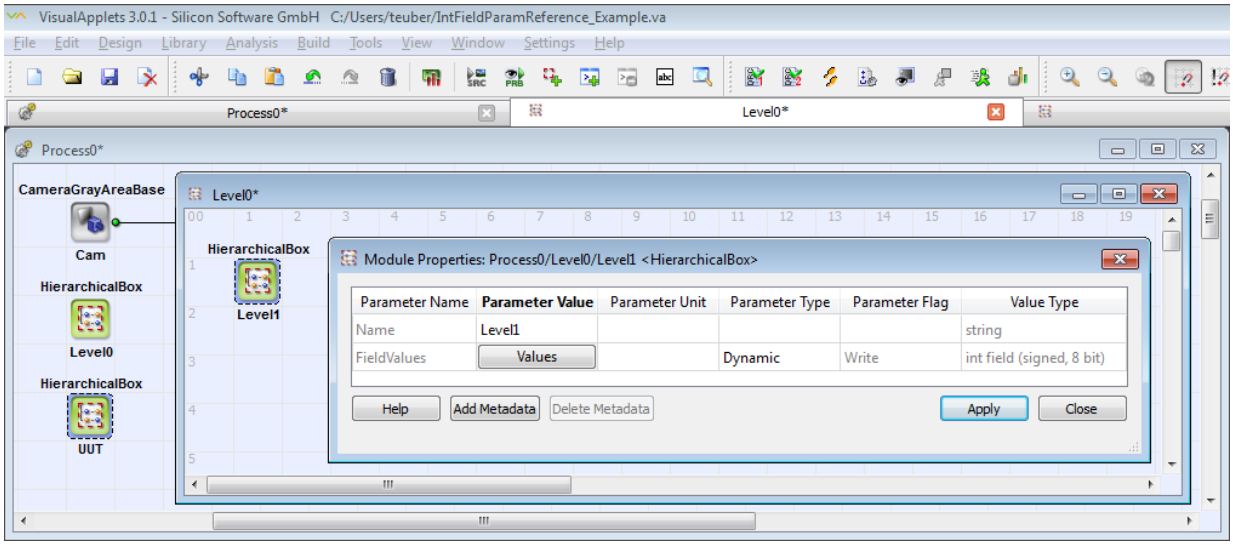
The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

Example:




The map parameter `FieldValues` is now available in Process0/Level0/Level1:




## 28.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

### 28.8.2. Supported Link Format

None

### 28.8.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |

Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.

This parameter is not part of the final applet.

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

If you want to grant access from and to the referenced field parameter (defined in parameter *Reference*) at another point in the design (in addition to the reference operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.

If this field is empty, no new parameter is created.

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus,

| DisplayHierarchy |
|---|
| parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.<br><br><HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.<br><br>Example: `Level0/Level1` |

| Index | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | UInt |
| Set the field index here. | |

| Value | |
|---|---|
| **Type** | dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit), unsigned integer (64 bit) |
| Set or read the field entry addressed by parameter *Index*. | |

| Field | |
|---|---|
| **Type** | static,dynamic, write, read parameter |
| **Default** | none |
| **Range** | array |
| Map parameter for access to referenced field parameter. This parameter mirrors all properties of the referenced field parameter. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the parameter *Value*. | |

| Reference | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string defining the referenced parameter: relative path to module, module name, and parameter name |

Here, you define the position of the referenced parameter within the design.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.



**Syntax for Referencing**

The syntax is as follows:

| Reference |
| --- |
| {<Path>/}<Module>.<ParamName><br><br><Path> is the relative path to the hierarchical level where the referenced module "Module" is located. Use a slash as separator between different hierarchical levels. Define the path as relative path **from the hierarchical level the reference operator instance is located in**.<br><br><Module> is the name of the module (operator instance or hierarchical box) that contains the referenced parameter.<br><br><ParamName> is the actual name of the referenced parameter.<br><br>If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/LUT.LUTContent`).<br><br>You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules. |

| Activate | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameter (via map parameter *Value*) is activated.<br><br>*No* = Access to and from referenced parameter is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.8.4. Examples of Use

The use of operator IntFieldParamReference is shown in the following examples:

- Section 14.1, 'Parameter Redirection'

  Examples - Demonstration how to use the parameter reference operators.

## 28.9. Operator IntParamReference

**Operator Library: Parameters**

This operator generates a 1:1 map parameter (in parameter *Value*) out of a module parameter located anywhere in the design. You specify the referenced parameter (path and name) in parameter *Reference*.



**Availability**

To use the *IntParamReference* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced parameter can be of one of the three following types: VA_SINT, VA_UINT or VA_ENUM. The map parameter itself is always of type Int64. The map parameter mirrors all properties of the referenced parameter.

If you want to reference an unsigned parameter that uses the full 64 bit, a re-interpretation of the value according to type Int64 is necessary.

You can define a target within the design hierarchy where the referenced parameter will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the reference operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the reference operator itself is located in).

This allows you to make a parameter of your design available on any hierarchical level (of the design) you want. For example, you can make the parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



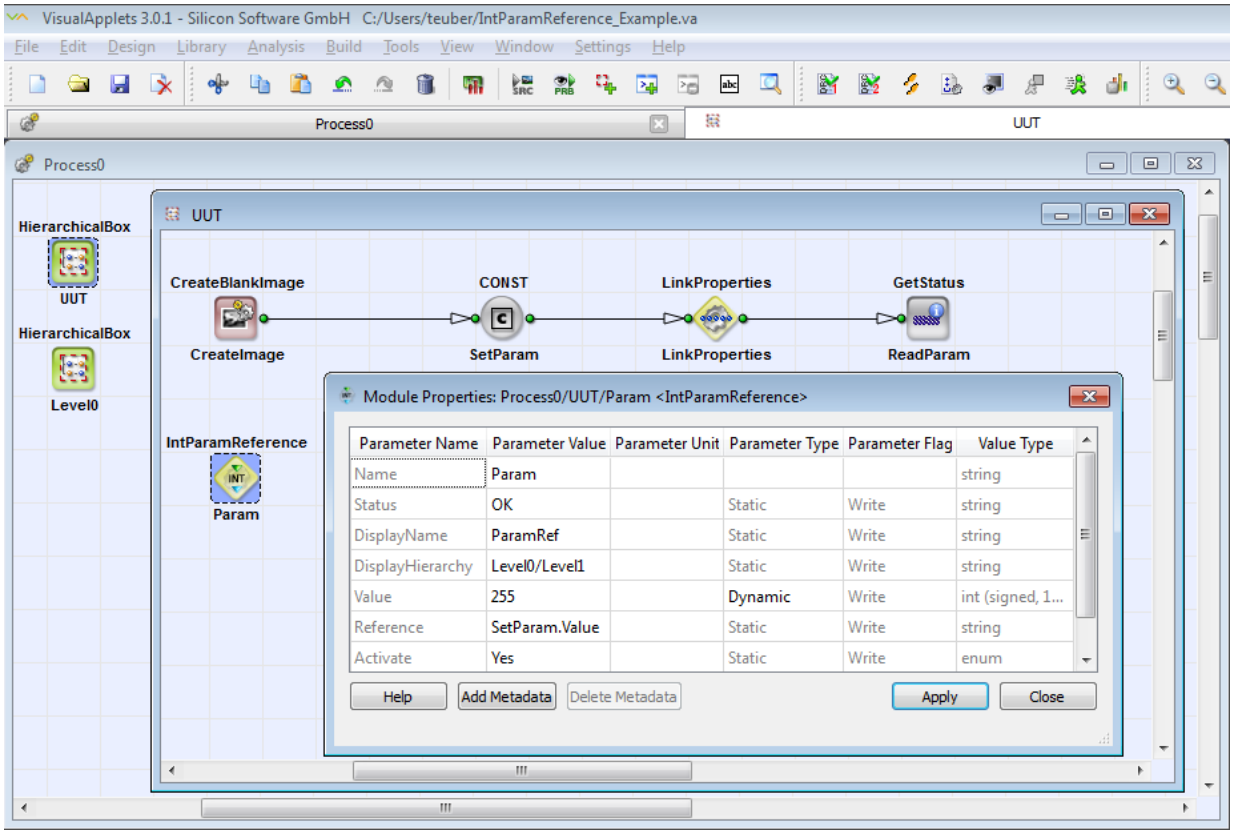
**All connected parameters update at once**

If one of the connected parameters is changed (referenced parameter, map parameter *Value*, or target parameter), the value is changed in all other connected parameters, too.

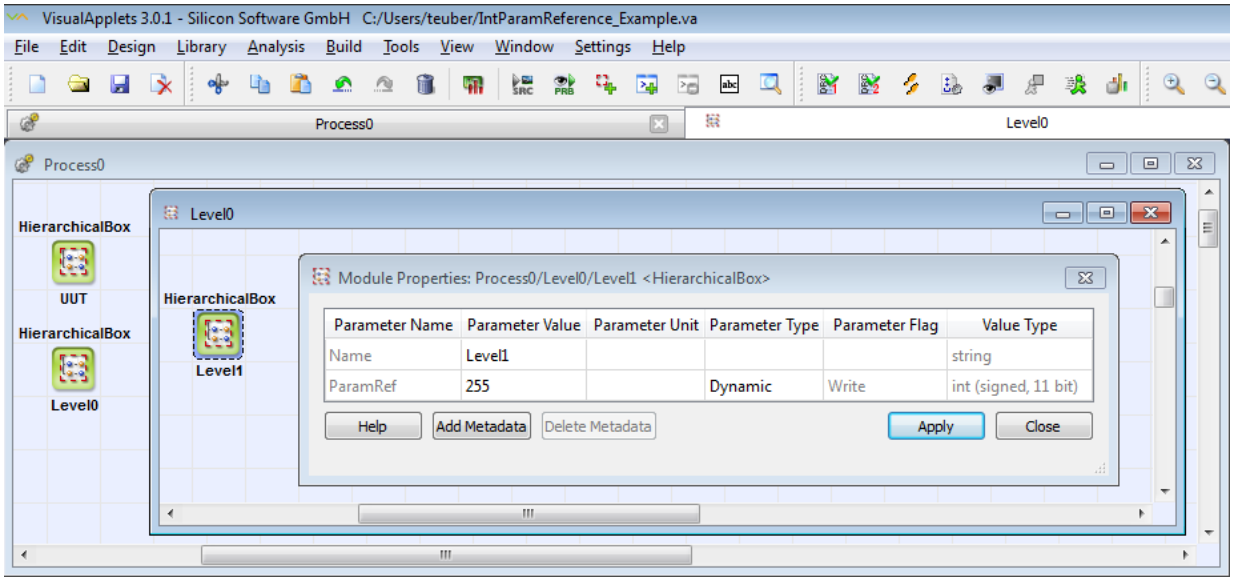
The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

Example:




Map parameter ParamRef is now available in Process0/Level0/Level1:




## 28.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

### 28.9.2. Supported Link Format

None

### 28.9.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |

Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.

This parameter is not part of the final applet.

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

If you want to grant access from and to the referenced parameter (defined in parameter *Reference*) at another point in the design (in addition to the reference operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.

If this field is empty, no new parameter is created.

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus,

| DisplayHierarchy |
|---|
| parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.<br><br><HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.<br><br>Example: `Level0/Level1` |

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit), unsigned integer (64 bit) |
| Map parameter for access from and to the referenced parameter. This parameter has the same properties as the referenced parameter. | |

| Unit | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |
| Enter here the unit for the parameter *Value*. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the parameter *Value*. | |

| Reference | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string defining the referenced parameter: relative path to module, module name, and parameter name |

Here, you define the position of the referenced parameter within the design.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.



**Syntax for Referencing**

The syntax is as follows:

```
{<Path>/}<Module>.<ParamName>
```

<Path> is the relative path to the hierarchical level where the referenced module "Module" is located. Use a slash as separator between different hierarchical levels. Define the path as relative path **from the hierarchical level the reference operator instance is located in**.

| Reference | |
|---|---|
| | <Module> is the name of the module (operator instance or hierarchical box) that contains the referenced parameter.<br><br><ParamName> is the actual name of the referenced parameter.<br><br>If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/SetParam.Value`).<br><br>You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules. |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameter (via map parameter *Value*) is activated.<br><br>*No* = Access to and from referenced parameter is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.9.4. Examples of Use

The use of operator IntParamReference is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 14.1, 'Parameter Redirection'

  Examples - Demonstration how to use the parameter reference operators.

- Section 14.3, 'User Library Parameter'

  Examples - Demonstration how user library elements can be provided with parameters.

# 28.10. Operator IntParamTranslator

**Operator Library: Parameters**

Operator *IntParamTranslator* allows translated access (read and write) to parameters of one or several other operators. For calculation, you define formulas. The syntax for these formulas complies to GenICam API version 2.0. A detailed syntax description you find below.

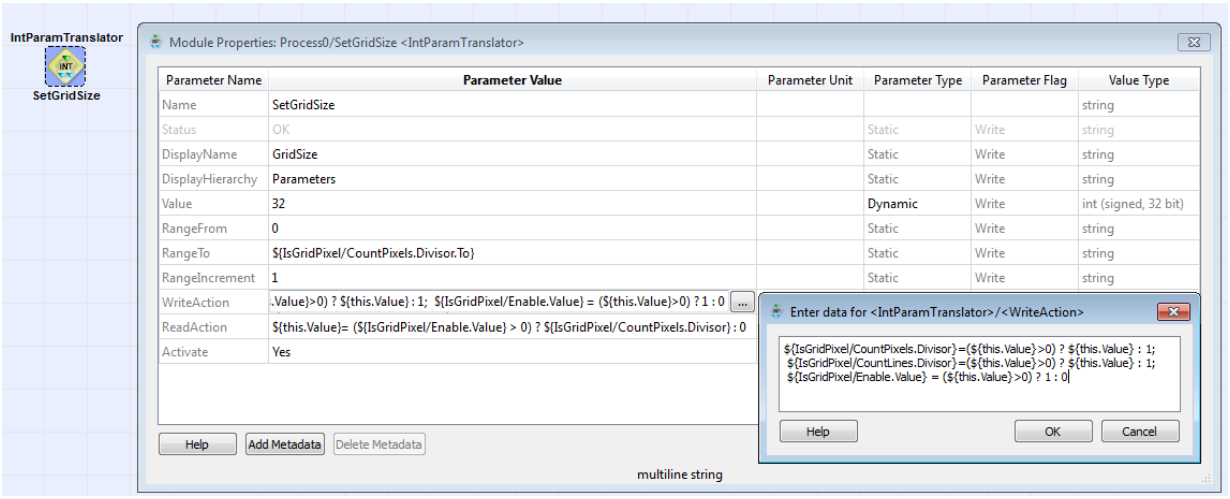


**Availability**

To use the *IntParamTranslator* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

Operator *IntParamTranslator* generates a map parameter of type VA_SINT. The name of the map parameter is *Value*.

Although the operator defines the type of its parameter *Value* to be of type VA_SINT, it does not restrict the data types of the parameters which are accessed during the write or read operations. For example, the formulas defined in the operator can contain access to parameters of type float.




**Read Action**

In parameter *ReadAction*, you define what happens for a read access to parameter *Value* of the translation operator. Parameter *Value* doesn't simply hold a value. Instead, the value of parameter *Value* is calculated out of a formula. This formula you define in parameter *ReadAction*. As soon as there happens a read access to parameter *Value*, the formula you defined in *ReadAction* is carried out and the result is forwarded to the element asking for the value.

The formula can contain values of operator parameters anywhere in the design. As soon as one of these values changes, the calculation of translation operator's parameter *Value* will end up with a new result (different to the value parameter *Value* held before). The formula can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.

**Write Action**

In parameter *WriteAction*, you define what happens during a write access to parameter *Value* of the translation operator. In contrast to using the reference operators, you do not simply forward the value of parameter *Value* to another parameter in the design. Instead, the value of parameter *Value* is used for a set of write actions you define in parameter *WriteAction* of the translation operator. Write actions are composed of one or several equations. On the left-hand side of these equations you define which parameter of which operator receives the result of the calculation. If you define more than one equation, separate the equations via semicolon. As soon as the there happens a write access to parameter *Value*, the formula(s) on the right-hand side of the equations you defined in parameter *WriteAction* is/are carried out and the result(s) is/are forwarded to the parameters specified on the left-hand side of the equations.

The formula can refer to values of operator parameters anywhere in the design. As soon as one of these values changes, the calculation results will differ when you re-write the same value as before to the *Value* parameter. The formula can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.

**Formulas for Calculation of Value Range**

You can also add formulas to calculate the current value range of parameter *Value*. You define the according formulas in the parameters *RangeFrom*, *RangeTo*, and *RangeIncrement* (step size). The formulas can contain values of module parameters anywhere in the design, as well as specific properties of these parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.



### Formula Syntax: Mathematical Operations

The syntax complies to the GenICam API standard in version 2.0. The allowed formula elements are identical to the formula elements defined in the GenICam standard:

| **Basic operations:** | |
|---|---|
| () | brackets |
| + - * / | addition, subtraction, multiplication, division |
| % | remainder |
| ** | power |
| & \| ^ ~ | bitwise: and / or / xor / not |
| <> = > < <= >= | logical relations: not equal / equal / greater / less / less or equal / greater or equal |
| && \|\| | logical and / logical or |
| << >> | shift left / shift right |

Table 28.5. Basic operations

**Conditional operator**

<condition> **?** <true_expr> **:** <false_expr>

Example:

${target.Value} = (${this.Value} > 0) ? 1 : 0;

| **Functions:** | |
|---|---|
| SGN(x) | return sign of x. Returns +1 for positive argument and -1 for negative argument |
| NEG(x) | swap sign of x |
| ABS(x) | return absolute value of x |
| SQRT(x) | return square root of x |
| TRUNC(x) | truncate x, which means returning the nearest integral value that is not larger in magnitude than x |
| FLOOR(x) | Round downward, returning the largest integral value that is not greater than x |

| | |
|---|---|
| CEIL(x) | Round upward, returning the smallest integral value that is not less than x |
| ROUND(x,precision) | round x to the number of decimal fractional digits given by precision, with halfway cases rounded away from zero |
| SIN(x) | return sine of an angle of x radians |
| COS(x) | return cosine of an angle of x radians |
| TAN(x) | return the tangent of an angle of x radians |
| ASIN(x) | return the principal value of the arc sine of x, expressed in radians |
| ACOS(x) | return the principal value of the arc cosine of x, expressed in radians |
| ATAN(x) | return the principal value of the arc tangent of x, expressed in radians |
| EXP(x) | return the base-e exponential function of x, which is e raised to the power x: ex |
| LN(x) | return the natural logarithm of x The natural logarithm is the base-e logarithm: the inverse of the natural exponential function (exp). |
| LG(x) | return the common (base-10) logarithm of x |
| E() | return Euler's number, 2.7182818284590451 |
| PI() | return circle constant, 3.1415926535897931 |

Table 28.6. Functions

Example:

${target.Value} = NEG(${this.Value})



**GenICam Support for Individual Functions**

With access via GenICam, support for funtions SGN(x) and NEG(x) is guaranteed. The other functions mentioned above may or may not be supported by a specific GenICam implementation. This is of special interest to eVA platforms using the GenICam interface.



**Paths to Parameters**

To access an operator parameter any place within your design, you need to provide the path to this parameter in your formula.

For access to an operator's parameter, you use the following construct:

```
${PathToModule/Module.ParamName}
```

"PathToModule": Here, you define the relative path to the operator who's parameter you want to access. The path is relative to the hierarchical level the translation operator itself is located. You also define the name of the accessed operator. Use a slash as hierarchy separator.

"Module": Name of module.

As name for the translation operator instance itself you use the name "this".



**Keep your Modules Independent**

It is not allowed to define a path towards a hierarchical level higher than the hierarchical level the translation operator is located. This rule follows the logic that a hierarchical module is not allowed to know anything about the environment it is instantiated in, because only in this case it can be used as a freely relocatable and replicable module.



**Access to Parameter Properties**

The formulas can not only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, or step size:

- `${PathToModule/Module.ParamName.From}` or `${PathToModule/Module.ParamName.Min}`: minimal valid value of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.To}` or `${PathToModule/Module.ParamName.Max}`: maximum valid value of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Inc}`: Increment (step size) between two valid values of parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Enum("EnumName")}`: Integer value of enumeration name `EnumName`.



**Syntax for Write Access Equations**

Equations for write actions you define in parameter WriteAction. They have the following syntax:

```
${PathToTargetModule/TargetModule.TargetParamName}=GenICamFormula
```

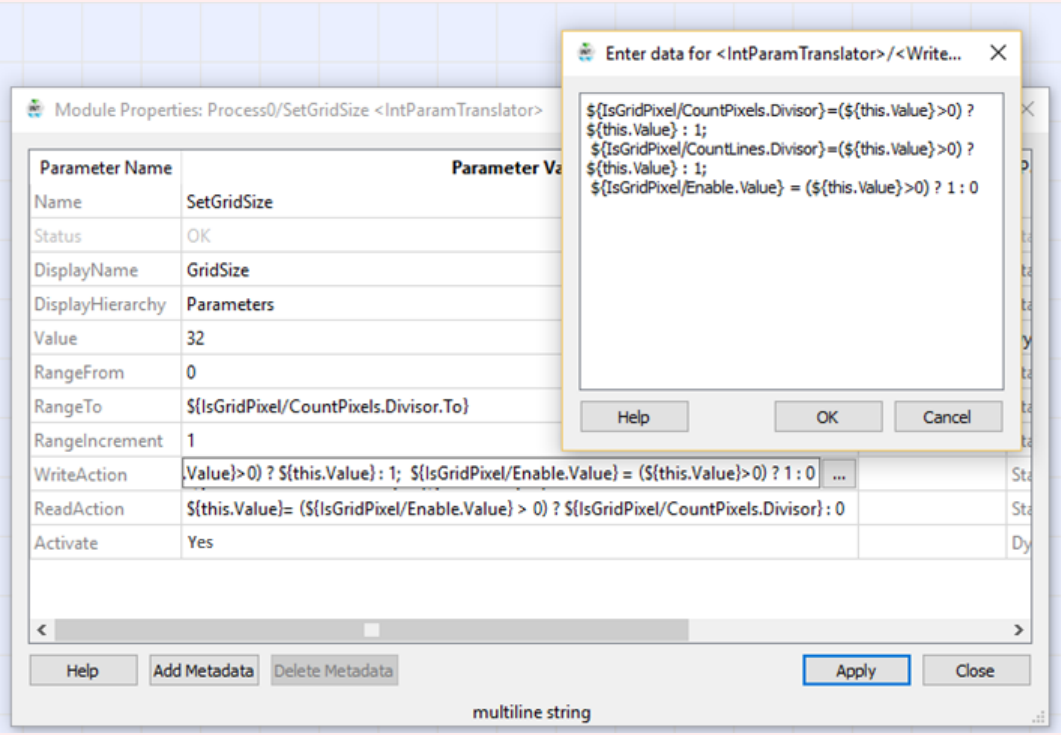
Here, you can define multiple equations for multiple target parameters. Use a semicolon as separator between the individual equations.

"this" refers to the translation operator instance itself.

Example for a WriteAction Equation:

```
${SetValue.Value}=${this.Value}+1
```

Example:






### Syntax for Read Access Equation

The equation for the read action you define in parameter *ReadAction*. The equation always starts with "${this.Value}=" . It has the following syntax:

```
${this.Value}=GenICamFormula
```

Example for a ReadAction equation:

```
${this.Value}=${SetValue.Value}-1
```

### Referencing Map Parameters

You can define a target within the design hierarchy where map parameter *Value* will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the translation operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the translation operator itself is located in).

This allows you to make map parameter *Value* accessible on any hierarchical level (of the design) you want. This is especially helpful when workin with protected hierarchical boxes. You can make the parameter available directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a translation operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

### 28.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

### 28.10.2. Supported Link Format

None

### 28.10.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |
| Displays the error status. If parameter *Activate* is set to *Yes*, the other operator settings are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.<br><br>This parameter is not part of the final applet. | |

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| If you want to grant access from and to the parameter *Value* at another point in the design, you need to define a new parameter. Specify here the name of this new parameter.<br><br>If this field is empty, no new parameter is created. | |

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.<br><br>If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.) | |

| DisplayHierarchy |
|---|



### Syntax for Setting up Path to Display Target

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit) |
| Parameter for access to other parameters via the equations specified for read and write accesses. | |

| Unit | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |
| Enter here the unit for the parameter *Value*. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the parameter *Value*. | |

| RangeFrom | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | |
| Definition of smallest valid value. You can also enter a formula here. | |

| RangeTo | |
|---|---|
| **Type** | static write parameter |
| **Default** | 255 |
| **Range** | |
| Definition of biggest valid value. You can also enter a formula here. | |

| RangeIncrement | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Definition of intervall between valid values (definition of stepsize). You can also enter a formula here. | |

| WriteAction | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | |
| Definition of equations for translation during write access to parameter *Value*.<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting.<br>To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and type "${". More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'. | |

| ReadAction | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | |
| Definition of equation for translation during read access to parameter *Value*.<br><br>The equation always starts with "${this.Value}=" .<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting.<br>To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and type "${". More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'. | |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameters (via read and write equations) is activated.<br><br>*No* = Access to and from referenced parameters (via read and write equations) is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.10.4. Examples of Use

The use of operator IntParamTranslator is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 14.2, 'Parameter Translation'

  Examples - Demonstration how to use the parameter translation operators for manipulation of parameters.

# 28.11. Operator IntVariable

**Operator Library: Parameters**

Operator *IntVariable* generates a software variable.



**Availability**

To use the *IntVariable* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

You can define a target within the design hierarchy where the variable will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the operator itself is located in).

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

## 28.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

## 28.11.2. Supported Link Format

None

## 28.11.3. Parameters

| **Status** | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |
| Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.<br><br>This parameter is not part of the final applet. | |

| **DisplayName** | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| If you want to grant access from and to the parameter at another point in the design (in addition to the operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.<br><br>If this field is empty, no new parameter is created. | |

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the operator instance. (Keep in mind this is not possible if the operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit), unsigned integer (64 bit) |

Map parameter for access from and to the variable. This parameter mirrors all properties of the variable.

| Unit | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |

Enter here the unit for the parameter *Value*.

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |

| Description |
|---|
| Enter here the description for the parameter *Value*. |

| RangeFrom | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | |
| Definition of smallest valid value. You can also enter a formula here. | |

| RangeTo | |
|---|---|
| **Type** | static write parameter |
| **Default** | 255 |
| **Range** | |
| Definition of biggest valid value. You can also enter a formula here. | |

| RangeIncrement | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Definition of intervall between valid values (definition of stepsize). You can also enter a formula here. | |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from the variable (via map parameter *Value*) is activated.<br><br>*No* = Access to and from the variable is de-activated and parameter *Status* is disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

# 28.12. Operator IntFieldVariable

**Operator Library: Parameters**

The *IntFieldVariable* operator generates a software variable field.



**Availability**

To use the *IntFieldVariable* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

You can define a target within the design hierarchy where the variable will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the operator itself is located in. With the *DisplayName* parameter, you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the operator itself is located in).

The operator has no inputs and outputs, because it doesn't interfere with the data flow structure of the design.

For a general introduction into the **Parameters** library, see 28. *Library Parameters* [1049].

## 28.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None, since there are no inputs or outputs. |

## 28.12.2. Supported Link Format

None

## 28.12.3. Parameters

| **Status** | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |
| Displays the error status. If the *Activate* parameter is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e. either *OK* or an error message.<br><br>This parameter is not part of the final applet. | |

| **DisplayName** | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |
| If you want to grant access from and to the parameter at another point in the design (in addition to the operator instance itself), you need to define a new parameter. Specify here the name of this new parameter.<br><br>If this field is empty, no new parameter is created. | |

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined with the *DisplayName* parameter.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new *DisplayName* parameter, but leave the *DisplayHierarchy* parameter empty: The new parameter is added to the hierarchical box that contains the operator instance. This is not possible if the operator instance is located on the highest (process) level.



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in the *DisplayHierarchy* parameter.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from the process level** to the level of the design where the <Hierarchical Box> is located which will get the new parameter. Thus, the *DisplayHierarchy* parameter can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

Example: `Level0/Level1`

| Size | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Unsigned integer (31 bit) |

With this parameter you can define the number of field entries.

| ColumnCount | |
|---|---|
| **Type** | static write, read parameter |
| **Default** | 0 |
| **Range** | [1,256] |

With this parameter you can define the number of columns for entering field values in the module properties dialog.

| Index | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | UInt |

| **Index** |
|---|
| With this parameter you can set the field index here. |

| **Value** | |
|---|---|
| **Type** | dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit) |
| With this parameter you can set or read the field entry addressed by the *Index* parameter. | |

| **Field** | |
|---|---|
| **Type** | static,dynamic, write, read parameter |
| **Default** | none |
| **Range** | array |
| This field parameter defines the content of the variable field. | |

| **Unit** | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |
| Enter here the unit for the *Value* parameter. | |

| **Description** | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | any text |
| Enter here the description for the *Value* parameter. | |

| **RangeFrom** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | |
| With this parameter you can define the smallest valid value. You can also enter a formula here. | |

| **RangeTo** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 255 |
| **Range** | |
| With this parameter you can define the biggest valid value. You can also enter a formula here. | |

| **RangeIncrement** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| With this parameter you can define the interval between valid values (definition of step size). You can also enter a formula here. | |

| **Activate** | |
|---|---|
| **Type** | dynamic write parameter |

| **Activate** | |
|---|---|
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from the variable is activated via the *Value* map parameter.<br><br>*No* = Access to and from the variable is de-activated and the *Status* parameter is disabled. The *DisplayName* and *DisplayHierarchy* parameters have no effect. | |

# 28.13. Operator LinkProperties

**Operator Library: Parameters**

This operator offers read access to the properties of the connected link. The link properties are available as read-only operator parameters.

The benefit of this operator is that the reference and translate operators of the *Parameters* library can use the link parameters for incorporating them into formulas. For example, in some formulas you may need to incorporate the link parallelism; by using this operator, the parallelism can be detected. This way, the formulas can automatically adapt to the current link configuration.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].



**Availability**

To use the *LinkProperties* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

## 28.13.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, Image data input |
| **Output Link** | O, Image data output |

## 28.13.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 28.13.3. Parameters

| Protocol | |
|---|---|
| **Type** | static write parameter |
| **Default** | VALT_IMAGE2D |
| **Range** | {VALT_IMAGE2D,VALT_LINE1D,VALT_PIXEL0D,VALT_SIGNAL} |
| Allows access to link property *Protocol*. Possible values are VALT_IMAGE2D,VALT_LINE1D,VALT_PIXEL0D,VALT_SIGNAL. | |

| ColorFormat | |
|---|---|
| **Type** | static write parameter |
| **Default** | VAF_GRAY |
| **Range** | {VAF_GRAY,VAF_COLOR,VAF_NONE,VAF_UNDEFINED} |
| Allows access to link property *ColorFormat*. Possible values are VAF_GRAY,VAF_COLOR,VAF_NONE,VAF_UNDEFINED. | |

| ColorFlavor | |
|---|---|
| **Type** | static write parameter |
| **Default** | FL_NONE |
| **Range** | {FL_NONE,FL_RGB,FL_HSI,FL_YUV,FL_LAB,FL_XYZ,FL_HSL,FL_HSV,FL_YCrCb} |
| Allows access to link property *ColorFlavor*. Possible values are FL_NONE, FL_RGB, FL_HSI, FL_YUV, FL_LAB, FL_XYZ, FL_HSL, FL_HSV, FL_YCrCb. | |

| Arithmetic | |
|---|---|
| **Type** | static write parameter |
| **Default** | UNSIGNED |
| **Range** | {SIGNED,UNSIGNED} |
| Allows access to link property *Arithmetic*. Possible values are SIGNED and UNSIGNED. | |

| BitWidth | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Allows access to link property *BitWidth*. | |

| Parallelism | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Allows access to link property *Parallelism*. | |

| KernelRows | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Allows access to link property *KernelRows*. | |

| KernelColumns | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Allows access to link property *KernelColumns*. | |

| MaxImgWidth | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1024 |
| **Range** | |

| **MaxImgWidth** |
| --- |
| Allows access to link property *MaxImgWidth*. |

| **MaxImgHeight** | |
| --- | --- |
| **Type** | static write parameter |
| **Default** | 1024 |
| **Range** | |
| Allows access to link property *MaxImgHeight*. | |

## 28.14. Operator LinkParamTranslator

**Operator Library: Parameters**

This operator offers read or write access to the properties of the connected link. Write access may trigger secondary parameter updates via write actions.

The benefit of this operator is that it enables changing link properties via other operators of the library *Parameters*. Additionally, you may use this operator to change other module parameters when a link parameter changes. For that purpose you can define write actions. You can also use this operator to fix link properties and enforce dependencies between several link properties.



**Availability**

To use the *LinkParamTranslator* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The link format of the output *O* strictly follows the format of the input *I*. The module parameters reflect the current status of the link properties at the port *I*. You may change the module parameters from *Read* (default) to *Write*. The mode *Read* means that the corresponding link property is read-only. This way the operator performs like the operator *LinkProperties*. If a parameter is set to *Write*, this parameter can be set. In that case, the parameter value is propagated to the corresponding link property. This has the additional effect, that the corresponding link property at the output port *O* cannot be edited any more, which is useful for selectively freezing link properties. Additionally, the corresponding link property from an input connection is not propagated to the property of *I* and *O* any more. In *Write* mode, the link property parameter can be edited even when the same property of a connected input link cannot be edited. In that case an error on the input link may show up indicating that the format of an attached previous module doesn't match to the format of the *LinkParamTranslator* instance.

In the parameter *WriteAction* you define what happens when link properties change. The parameter *Activate* activates the write actions defined in *WriteAction*:

- If *Activate* is set to *No*, then parameters which have the access flag *Write* may still be changed but no write action is executed. As any link property is a static parameter, the possible write targets must be static, too.
- If the write target is a parameter of the own module, then the addressed module parameter may have the mode *Read* or *Write*.
- If the parameter has the mode *Read*, the write action may be used to enforce certain rules on the link properties.

Write actions are composed of one or several equations. If you define more than one equation, separate the equations via semicolon. On the left-hand side of these equations you define which parameter of which module receives the result of the calculation. Use the notation `this` when referencing a parameter of this module. As soon as the link properties are changed either because of an update of the incoming link or write access to a parameter of this module, the formula(s) on the right-hand side of the equation(s) you defined in parameter *WriteAction* is/are carried out and the result(s) is/are forwarded to the parameter(s) specified on the left-hand side of the equation(s). As with the left-hand side you can reference parameters of this module using the notation `this`.

Example:

```
${this.MaxImgWidth} = (${this.MaxImgWidth} <= 1024) ? ${this.MaxImgWidth} : 1024;
```

The formula can refer to values of static module parameters anywhere in the design. The formula cannot only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, step size, or the numerical value of enumeration items.



**Formula syntax: mathematical operations**

The syntax complies to the GenICam API standard in version 2.0. The allowed formula elements are identical with the formula elements defined in the GenICam standard:

| **Basic operations:** | |
|---|---|
| () | Brackets |
| + - * / | Addition, subtraction, multiplication, division |
| % | Remainder |
| ** | Power |
| & \| ^ ~ | Bitwise: and / or / xor / not |
| <> = > < <= >= | Logical relations: not equal / equal / greater / less / less or equal / greater or equal |
| && \|\| | Logical and / logical or |
| << >> | Shift left / shift right |

Table 28.7. Basic operations

**Conditional operator**

<condition> **?** <true_expr> **:** <false_expr>

Example:

${target.Value} = (${this.BitWidth} > 8) ? 2 : 1;

| **Functions:** | |
|---|---|
| SGN(x) | Returns sign of x. Returns +1 for positive argument and -1 for negative argument. |
| NEG(x) | Swaps sign of x. |
| ABS(x) | Returns absolute value of x. |
| SQRT(x) | Returns square root of x. |
| TRUNC(x) | Truncates x, which means returning the nearest integral value that is not larger in magnitude than x. |
| FLOOR(x) | Rounds downward, returning the largest integral value that is not greater than x. |
| CEIL(x) | Rounds upward, returning the smallest integral value that is not less than x. |
| ROUND(x,precision) | Rounds x to the number of decimal fractional digits given by precision, with halfway cases rounded away from zero. |
| SIN(x) | Returns sine of an angle of x radians. |
| COS(x) | Returns cosine of an angle of x radians. |
| TAN(x) | Returns the tangent of an angle of x radians. |
| ASIN(x) | Returns the principal value of the arc sine of x, expressed in radians. |
| ACOS(x) | Returns the principal value of the arc cosine of x, expressed in radians. |
| ATAN(x) | Returns the principal value of the arc tangent of x, expressed in radians. |

| | |
|---|---|
| EXP(x) | Returns the base-e exponential function of x, which is e raised to the power x: ex. |
| LN(x) | Returns the natural logarithm of x. The natural logarithm is the base-e logarithm: the inverse of the natural exponential function (exp). |
| LG(x) | Returns the common (base-10) logarithm of x. |
| E() | Returns Euler's number, 2.7182818284590451. |
| PI() | Returns circle constant, 3.1415926535897931. |

Table 28.8. Functions

Example:

${target.Value} = NEG(${this.Parallelism})



### Paths to parameters

To access an operator parameter any place within your design, you need to provide the path to this parameter in your formula.

For access to an operator's parameter, use the following construct:

```
${PathToModule/Module.ParamName}
```

`PathToModule`: Here, you define the relative path to the operator whose parameter you want to access. The path is relative to the hierarchical level the translation operator itself is located. You also define the name of the accessed operator. Use a slash as hierarchy separator.

`Module`: Name of the module.

As name for the translation operator instance itself, use the name `this`.



#### Keep your modules independent

It is not allowed to define a path towards a hierarchical level higher than the hierarchical level the translation operator is located at. This rule follows the logic that a hierarchical module is not allowed to know anything about the environment it is instantiated in, because only in this case it can be used as a freely relocatable and replicable module.



### Access to parameter properties

The formulas cannot only incorporate the values of parameters, but also specific properties of parameters, such as minimal value, maximal value, or step size:

- `${PathToModule/Module.ParamName.From}` or `${PathToModule/Module.ParamName.Min}`: the minimal valid value of the parameter `PathToModule/Module.ParamName`.

- `${PathToModule/Module.ParamName.To}` or `${PathToModule/Module.ParamName.Max}`: the maximum valid value of the parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Inc}`: increment (step size) between two valid values of the parameter `PathToModule/Module.ParamName`.
- `${PathToModule/Module.ParamName.Enum("EnumName")}`: integer value of the enumeration name `EnumName`.



**Syntax for write access equations**

You define the equations for write actions in the parameter *WriteAction*. They have the following syntax:

```
${PathToTargetModule/TargetModule.TargetParamName}=GenICamFormula
```

Here, you can define multiple equations for multiple target parameters. Use a semicolon as separator between the individual equations.

`this` refers to the translation operator instance itself.

You find a general introduction into the library `Parameters` in 28. *Library Parameters* [1049].

### 28.14.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, Image data input |
| **Output Link** | O, Image data output |

### 28.14.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | As I |
| **Arithmetic** | {Unsigned, signed} | As I |
| **Parallelism** | Any | As I |
| **Kernel Columns** | Any | As I |
| **Kernel Rows** | Any | As I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | As I |
| **Color Format** | Any | As I |
| **Color Flavor** | Any | As I |
| **Max. Img Width** | Any | As I |
| **Max. Img Height** | Any | As I |

❶ The range of the input bit width is [1, 64]. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 28.14.3. Parameters

| **Status** | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | None |
| **Range** | 0K or an error message occurs |
| Displays the error status. If the parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message occurs.<br><br>This parameter is not part of the final applet. | |

| **Protocol** | |
|---|---|
| **Type** | static write parameter |
| **Default** | VALT_IMAGE2D |
| **Range** | {VALT_IMAGE2D,VALT_LINE1D,VALT_PIXEL0D,VALT_SIGNAL} |
| Allows access to the link property *Protocol*. Possible values are VALT_IMAGE2D,VALT_LINE1D,VALT_PIXEL0D,VALT_SIGNAL. | |

| **ColorFormat** | |
|---|---|
| **Type** | static write parameter |
| **Default** | VAF_GRAY |
| **Range** | {VAF_GRAY,VAF_COLOR,VAF_NONE,VAF_UNDEFINED} |
| Allows access to the link property *ColorFormat*. Possible values are VAF_GRAY,VAF_COLOR,VAF_NONE,VAF_UNDEFINED. | |

| **ColorFlavor** | |
|---|---|
| **Type** | static write parameter |
| **Default** | FL_NONE |
| **Range** | {FL_NONE,FL_RGB,FL_HSI,FL_YUV,FL_LAB,FL_XYZ,FL_HSL,FL_HSV,FL_YCrCb} |
| Allows access to the link property *ColorFlavor*. Possible values are FL_NONE, FL_RGB, FL_HSI, FL_YUV, FL_LAB, FL_XYZ, FL_HSL, FL_HSV, FL_YCrCb. | |

| **Arithmetic** | |
|---|---|
| **Type** | static write parameter |
| **Default** | UNSIGNED |
| **Range** | {SIGNED, UNSIGNED} |
| Allows access to the link property *Arithmetic*. Possible values are SIGNED and UNSIGNED. | |

| **BitWidth** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Allows access to the link property *BitWidth*. | |

| **Parallelism** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Allows access to the link property *Parallelism*. | |

| KernelRows | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Allows access to the link property *KernelRows*. | |

| KernelColumns | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | |
| Allows access to the link property *KernelColumns*. | |

| MaxImgWidth | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1024 |
| **Range** | |
| Allows access to the link property *MaxImgWidth*. | |

| MaxImgHeight | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1024 |
| **Range** | |
| Allows access to the link property *MaxImgHeight*. | |

| WriteAction | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | |

Definition of equations for translation during write access to a link property parameter.

For this parameter, you can use the autocompletion functionality and syntax highlighting.
To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and type "${". More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No, Yes} |

Yes = Access to referenced parameters via write equations in *WriteAction* is activated.

No = Access to referenced parameters via write equations is de-activated and the parameter *Status* is disabled.

## 28.14.4. Examples of Use

The use of operator LinkParamTranslator is shown in the following examples:

- Section 14.5, 'Link Parameter Translation'

  Examples - Demonstration of how to use the parameter translation operator LinkParamTranslator.

## 28.15. Operator StringParamReference

**Operator Library: Parameters**

This operator generates a 1:1 map parameter (in parameter "Value") out of a module parameter located anywhere in the design. You specify the referenced parameter (path and name) in parameter "Reference".



**Availability**

To use the *StringParamReference* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced parameter can be of one of the three following types: VA_STRING, VA_FILENAME, or VA_METADATA. The map parameter is always a string.

You can define a target within the design hierarchy where the referenced parameter will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the reference operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* will be available in the hierarchical box addressed by *DisplayHierarchy*. If *DisplayName* is empty, no parameter is created. If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e., the hierarchical box the reference operator itself is located in).

This allows you to make a parameter of your design available on any hierarchical level (of the design) you want. For example, you can make the parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



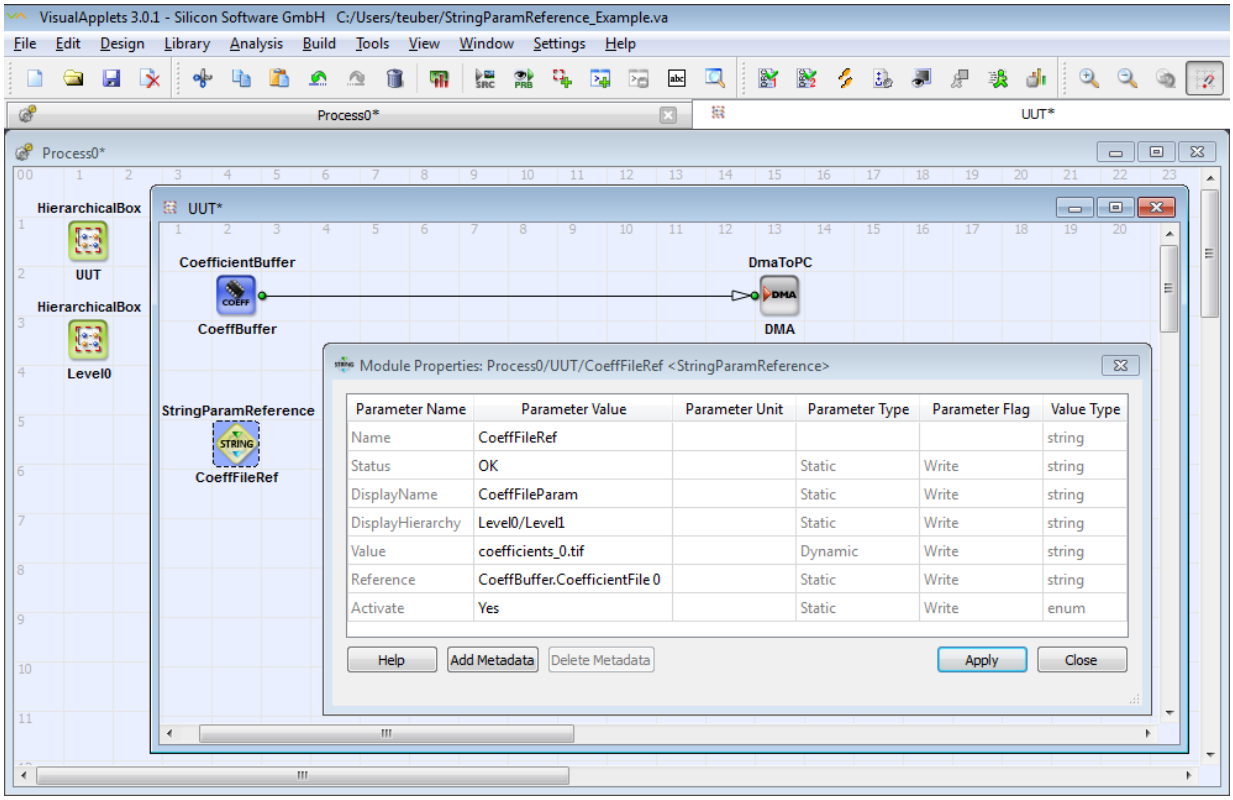
**All connected parameters update at once**

If one of the connected parameters is changed (referenced parameter, map parameter *Value*, or target parameter), the value is changed in all other connected parameters, too.

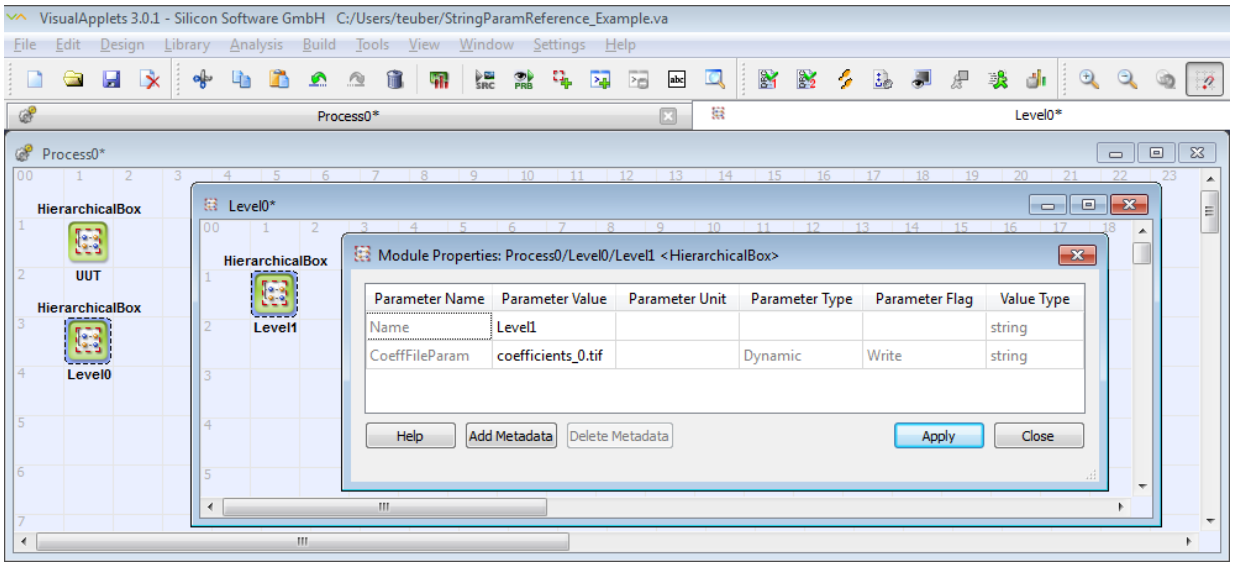
The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

A general introduction into library Parameters you find in 28. *Library Parameters* [1049].

Example:




Map parameter "CoeffFileParam" is now available in Process0/Level0/Level1:




## 28.15.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no inputs or outputs) |

## 28.15.2. Supported Link Format

None

### 28.15.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | none |
| **Range** | OK or error message |

Displays the error status. If parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message.

This parameter is not part of the final applet.

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

If you want to grant access from and to the referenced parameter (defined in parameter *Reference*) at another point in the design (in addition to the reference operator instance itself), you need to define a new parameter. Specify here the name of this new parameter. If this field is empty, no new parameter is created.

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. (Keep in mind this is not possible if the reference operator instance is located on the highest (process) level.)



**Syntax for Setting up Path to Display Target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

| DisplayHierarchy |
|---|
| Example: `Level0/Level1` |

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit), unsigned integer (64 bit) |
| Map parameter for access from and to the referenced parameter. This parameter has the same properties as the referenced parameter. | |

| Reference | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | any string defining the referenced parameter: relative path to module, module name, and parameter name |

Here, you define the position of the referenced parameter within the design.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.



### Syntax for Referencing

The syntax is as follows:

```
{<Path>/}<Module>.<ParamName>
```

<Path> is the relative path to the hierarchical level where the referenced module "Module" is located. Use a slash as separator between different hierarchical levels. Define the path as relative path **from the hierarchical level the reference operator instance is located in**.

<Module> is the name of the module (operator instance or hierarchical box) that contains the referenced parameter.

<ParamName> is the actual name of the referenced parameter.

If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/CoeffBuffer.CoefficientFile0`).

You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules.

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No,Yes} |
| *Yes* = Access to and from referenced parameter (via map parameter *Value*) is activated.<br><br>*No* = Access to and from referenced parameter is de-activated and parameter *Status* disabled. Parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

### 28.15.4. Examples of Use

The use of operator StringParamReference is shown in the following examples:

- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 28.16. Operator ResourceReference

**Operator Library: Parameters**

You find a general introduction into the library Parameters in 28. *Library Parameters* [1049].

This operator generates a 1:1 map parameter (in the parameter *ResourceIndex*) out of a device resource mapping for a module located anywhere in the design. You specify the referenced module in the parameter *Reference* and the device resource type in the parameter *ResourceName*.



**Availability**

To use the *ResourceReference* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced resource mapping is represented as an integer value.

You can define a target within the design hierarchy where the referenced resource mapping will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the rerence operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. This means:

- If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* is available in the hierarchical box addressed by *DisplayHierarchy*.
- If *DisplayName* is empty, no parameter is created.
- If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e. the hierarchical box the reference operator itself is located in).

This allows you to make the parameter for the resource mapping available on any hierarchical level of the design you want. For example, you can make the resource mapping of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped resource parameters can be referenced themselves. Thus, you can reference either the content of the parameter *ResourceIndex* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



**All connected parameters update at once**

If one of the connected parameters is changed (the referenced resource mapping, the map parameter *ResourceIndex*, or the display reference parameter), the value is changed in all other connected parameters, too.

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

For further details, refer to the general description of the the library Parameters in 28. *Library Parameters* [1049].

## 28.16.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None, since there are no inputs or outputs |

## 28.16.2. Supported Link Format

None

### 28.16.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | None |
| **Range** | 0K or an error message occurs |

Displays the error status. If parameter *Activate* is set to *Yes*, the other operator settings are checked. This parameter displays the result of this check, i.e., either *OK* or an error message occurs.

This parameter is not part of the final applet.

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | None |
| **Range** | Any string |

If you want to grant access from and to the referenced parameter defined in the parameter *Reference* at another point in the design in addition to the reference operator instance itself, you need to define a new parameter. Specify here the name of this new parameter. If this field is empty, no new parameter is created.

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | None |
| **Range** | Any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in the parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you defined a new parameter *DisplayName*, but leave parameter *DisplayHierarchy* empty: The new parameter is added to the hierarchical box that contains the reference operator instance. If the reference operator instance is located on the highest process level, this is not possible.



**Syntax for setting up the path to the display target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in the parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, the parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

| DisplayHierarchy |
|---|
| Example: `Level0/Level1` |

| ResourceIndex | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Depends on the index range of the connected device resource |
| Map parameter for access from and to the referenced device resource mapping value. | |

| ResourceName | |
|---|---|
| **Type** | static, dynamic, write parameter |
| **Default** | 0 |
| **Range** | Any string |
| Define the device resource type which shall be accessed. The name is the same as provided in the device resources dialog. | |

| Reference | |
|---|---|
| **Type** | static write parameter |
| **Default** | None |
| **Range** | Any string defining the referenced module: relative path to module |
| Here, you define the position of the referenced module within the design.<br><br>For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | None |
| **Range** | Any string |
| Here, you define a description of the generated reference parameter. | |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No, Yes} |
| *Yes* = Access to and from the referenced device resource mapping (via the map parameter *ResourceIndex*) is activated.<br><br>*No* = Access to and from the referenced device resource mapping is de-activated and the parameter *Status* is disabled. The parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.16.4. Examples of Use

The use of operator ResourceReference is shown in the following examples:

- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 28.17. Operator IntParamSelector

**Operator Library: Parameters**

This operator generates a map parameter in the parameter *Value*. You can switch this map parameter between several module parameters located anywhere in the design via the parameter *Select*. To specify path and name of the referenced parameters, add a list separated by semicolon in the parameter *References*.



**Availability**

To use the *IntParamSelector* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced parameters can be of one of the three following types:

- VA_SINT: signed integer like the parameter *Value* in the operator *CONST*.
- VA_UINT: unsigned integer like the parameter *ScaleFactor* in the operator *SCALE*.
- VA_ENUM: enumeration like the parameter *mode* in the operator *SetSignalStatus*.

The map parameter itself is always of type Int64. The map parameter mirrors all properties of the currently selected referenced parameter.

If you want to reference an unsigned parameter that uses the full 64 bit, a re-interpretation of the value according to type Int64 is necessary.

You can define a target within the design hierarchy where the selected parameter will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the reference operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. This means:

- If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* is available in the hierarchical box addressed by *DisplayHierarchy*.
- If *DisplayName* is empty, no parameter is created.
- If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e. the hierarchical box the reference operator itself is located in).

This allows you to make the selected parameter available on any hierarchical level of the design you want. For example, you can make the parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of the parameter *Value* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



**All connected parameters update at once**

If one of the connected parameters is changed (i.e. the selected parameter, the map parameter *Value*, or the display target parameter), the value is changed in all other connected parameters, too.

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

You find a general introduction into the library `Parameters` in 28. *Library Parameters* [1049].

## 28.17.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None, since there are no inputs or outputs |

### 28.17.2. Supported Link Format

None

### 28.17.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | None |
| **Range** | *OK* or an error message occurs |

Displays the error status. If the parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message occurs.

This parameter is not part of the final applet.

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | None |
| **Range** | Any string |

If you want to grant access from and to the referenced parameter defined in the parameter *Reference* at another point in the design in addition to the reference operator instance itself, you need to define a new parameter. Specify here the name of this new parameter. If this field is empty, no new parameter is created.

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | None |
| **Range** | Any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in the parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you define a new parameter *DisplayName*, but leave the parameter *DisplayHierarchy* empty, the new parameter is added to the hierarchical box that contains the reference operator instance. If the reference operator instance is located on the highest process level, this is not possible.



**Syntax for setting up the path to the display target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in the parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus,

| DisplayHierarchy |
|---|
| the parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.<br><br><HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.<br><br>Example: `Level0/Level1` |

| Select | |
|---|---|
| **Type** | static, dynamic, write parameter |
| **Default** | 0 |
| **Range** | Depends on the number of referenced parameters |
| Selector for switching between the referenced parameters. | |

| Value | |
|---|---|
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | Signed integer (64 bit), unsigned integer (64 bit) |
| Map parameter for access from and to the selected parameter. This parameter has the same properties as the currently selected referenced parameter. | |

| Unit | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |
| Enter here the unit for the parameter *Value*. | |

| Description | |
|---|---|
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Any text |
| Enter here the description for the parameter *Value*. | |

| References | |
|---|---|
| **Type** | static write parameter |
| **Default** | None |
| **Range** | String defining the referenced parameters in a list separated by semicolon: relative path to module, module name, and parameter name. |

Here, you define the position of the referenced parameters within the design.



**Syntax for referencing**

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

The syntax for an item in the string list is as follows:

| References |
|---|
| {<Path>/}<Module>.<ParamName><br><br><Path> is the relative path to the hierarchical level where the referenced module <Module> is located. Use a slash as separator between different hierarchical levels. Define the path as relative path **from the hierarchical level the reference operator instance is located in**.<br><br><Module> is the name of the module (i.e. an operator instance or a hierarchical box) that contains the referenced parameter.<br><br><ParamName> is the name of the referenced parameter.<br><br>If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/SetParam.Value`).<br><br>You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules. |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No, Yes} |
| *Yes* = Access to and from the referenced parameter (via the map parameter *Value*) is activated.<br><br>*No* = Access to and from the referenced parameter is de-activated and the parameter *Status* is disabled. The parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.17.4. Examples of Use

The use of operator IntParamSelector is shown in the following examples:

- Section 14.4, 'Parameter Selection'

  Examples - Demonstration of how to use the parameter translation operators *IntParamSelector* and *FloatParamSelector*.

# 28.18. Operator FloatParamSelector

**Operator Library: Parameters**

This operator generates a map parameter in the parameter *Value*. You can switch this map parameter between several module parameters located anywhere in the design via the parameter *Select*. To specify path and name of the referenced parameters, add a list separated by semicolon in the parameter *References*.



**Availability**

To use the *FloatParamSelector* operator, you need either an **Expert** license, a **Parameter Module** license, or the **VisualApplets 4** license.

The referenced parameters must be of type VA_DOUBLE. The map parameter mirrors all properties of the currently selected referenced parameter.

You can define a target within the design hierarchy where the selected parameter will be visible and accessible. The target is defined by two parameters: *DisplayHierarchy* and *DisplayName*. In *DisplayHierarchy* you address a hierarchical box within the same process the reference operator itself is located in. In parameter *DisplayName* you set up the name for the target parameter. This means:

- If *DisplayHierarchy* and *DisplayName* are both set, the parameter defined by *DisplayName* is available in the hierarchical box addressed by *DisplayHierarchy*.
- If *DisplayName* is empty, no parameter is created.
- If *DisplayName* is set, but *DisplayHierarchy* is empty, the parameter defined by *DisplayName* is added to the parent hierarchy (i.e. the hierarchical box the reference operator itself is located in).

This allows you to make the selected parameter available on any hierarchical level of the design you want. For example, you can make the parameters of modules within a protected hierarchical box available for parametrization directly under the properties of the protected hierarchical box itself.

Mapped parameters can be referenced themselves. Thus, you can reference either the content of parameter *Value* in a reference operator, or reference the parameter you have defined in the parameters *DisplayName* and *DisplayHierarchy*. This is especially important when protected hierarchical boxes are built up out of other protected hierarchical boxes.



**All connected parameters update at once**

If one of the connected parameters is changed (i.e. the selected parameter, the map parameter *Value*, or the display target parameter), the value is changed in all other connected parameters, too.

The operator has no inputs and outputs, as it doesn't interfere with the data flow structure of the design.

You find a general introduction into the library `Parameters` in 28. *Library Parameters* [1049].

## 28.18.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None, since there are no inputs or outputs |

## 28.18.2. Supported Link Format

None

### 28.18.3. Parameters

| Status | |
|---|---|
| **Type** | static read (although the GUI displays write) parameter |
| **Default** | None |
| **Range** | *OK* or an error message occurs |

Displays the error status. If the parameter *Activate* is set to *Yes*, the other module parameters are checked. This parameter displays the result of this check, i.e., either *OK* or an error message occurs.

This parameter is not part of the final applet.

| DisplayName | |
|---|---|
| **Type** | static write parameter |
| **Default** | none |
| **Range** | Any string |

If you want to grant access from and to the referenced parameter defined in the parameter *Reference* at another point in the design in addition to the reference operator instance itself, you need to define a new parameter. Specify here the name of this new parameter. If this field is empty, no new parameter is created.

| DisplayHierarchy | |
|---|---|
| **Type** | static write parameter |
| **Default** | None |
| **Range** | Any string |

Specify here the hierarchical box to which you want to attach the new map parameter you defined in the parameter *DisplayName*.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.

If you define a new parameter *DisplayName*, but leave the parameter *DisplayHierarchy* empty, the new parameter is added to the hierarchical box that contains the reference operator instance. If the reference operator instance is located on the highest process level, this is not possible.



**Syntax for setting up the path to the display target**

If you want the new parameter to show up in another place of the design, you need to specify the path to the hierarchical box that is to contain the new parameter in the parameter *DisplayHierarchy*.

The syntax is as follows:

```
{<Path>/}<HierarchicalBox>
```

<Path> is the relative path **from process level** to the hierarchical level of the design where the <Hierarchical Box> is located that is to receive the new parameter. Thus, the parameter *DisplayHierarchy* can only address hierarchical boxes within the same process. Use a slash as separator between different hierarchical levels.

<HierarchicalBox> is the name of the hierarchical box that is to receive the new map parameter.

| DisplayHierarchy |
| --- |
| Example: `Level0/Level1` |

| Select | |
| --- | --- |
| **Type** | static, dynamic, write parameter |
| **Default** | 0 |
| **Range** | Depends on number of referenced parameters |
| Selector for switching between the referenced parameters. | |

| Value | |
| --- | --- |
| **Type** | static, dynamic, write, read parameter |
| **Default** | 0 |
| **Range** | double |
| Map parameter for access from and to the selected parameter. This parameter has the same properties as the currently selected referenced parameter. | |

| Unit | |
| --- | --- |
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Latin-1 |
| Enter here the unit for the parameter *Value*. | |

| Description | |
| --- | --- |
| **Type** | static write parameter |
| **Default** | - |
| **Range** | Any text |
| Enter here the description for the parameter *Value*. | |

| References | |
| --- | --- |
| **Type** | static write parameter |
| **Default** | None |
| **Range** | String defining the referenced parameters in a list separated by semicolon: relative path to module, module name, and parameter name |

Here, you define the position of the referenced parameters within the design.

For this parameter, you can use the autocompletion functionality and syntax highlighting. To use autocompletion, click into the *Parameter Values* field of this parameter in the *Module Properties* dialog, and press the **TAB** key of your keyboard. More detailed instructions for the autocompletion functionality are available at Section 3.7.1.4, 'Autocompletion and Syntax Highlighting for Translator and Reference Operators'.



**Syntax for referencing**

The syntax for an item in the string list is as follows:

```
{<Path>/}<Module>.<ParamName>
```

<Path> is the relative path to the hierarchical level where the referenced module <Module> is located. Use a slash as separator between different hierarchical levels.

| References | |
|---|---|
| | Define the path as relative path **from the hierarchical level the reference operator instance is located in**.<br><br><Module> is the name of the module (i.e. an operator instance or a hierarchical box) that contains the referenced parameter.<br><br><ParamName> is the name of the referenced parameter.<br><br>If <Path> starts with a slash, the path is interpreted as absolute path starting from design level (e.g., `/Process0/UUT/SetParam.Value`).<br><br>You can also use `../` to go up a level. However, take care you don't use this option when designing freely replicable or relocatable modules. |

| Activate | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | No |
| **Range** | {No, Yes} |
| *Yes* = Access to and from referenced parameter (via map parameter *Value*) is activated.<br><br>*No* = Access to and from referenced parameter is de-activated and the parameter *Status* is disabled. The parameters *DisplayName* and *DisplayHierarchy* have no effect. | |

## 28.18.4. Examples of Use

The use of operator FloatParamSelector is shown in the following examples:

- Section 14.4, 'Parameter Selection'

  Examples - Demonstration of how to use the parameter translation operators *IntParamSelector* and *FloatParamSelector*.

# 29. Library Hardware Platform



The library *Hardware Platform* contains operators which are only available on certain hardware platform devices.

The following list summarizes all Operators of Library Hardware Platform

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | AppletProperties | Controls the applet and the board hardware setup. | Version 2.1 |
|  | BoardStatus | Provides read-only parameters to monitor the current board status during runtime. | Version 2.1 |
|  | ActionCommand | Sends an action command trigger signal to the firmware whereupon the firmware sends an action command to the camera. | Version 3.0.4 |
|  | CameraControl | Provides the interface for the Camera Link CC ports. | Version 1.3 |
|  | BaseGrayCamera | Supports any BASE GRAY configurations for line scan and area scan cameras. | Version 2.2 |
|  | BaseRgbCamera | Supports any CL RGB BASE configurations for line scan and area scan cameras. | Version 2.2 |
|  | MediumGrayCamera | Supports any MEDIUM GRAY configurations for line scan and area scan cameras. | Version 2.2 |
|  | MediumRgbCamera | Supports any CL RGB MEDIUM configurations for line scan and area scan cameras. | Version 2.2 |
|  | FullGrayCamera | Supports any FULL GRAY configurations for line scan and area scan cameras. | Version 2.1 |
|  | FullRgbCamera | Supports any CL RGB FULL configurations for line scan and area scan cameras. | Version 2.2 |
|  | CameraGrayArea | Camera image input interface for grayscale area scan GigE Vision cameras. | Version 1.2 SR5 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | CameraGrayAreaBase | Camera image input interface for grayscale area scan cameras in Camera Link base configuration mode. | Version 1.2 |
|  | CameraGrayAreaFull | Camera image input interface for grayscale area scan cameras in Camera Link full configuration mode. | Version 1.2 |
|  | CameraGrayAreaMedium | Camera image input interface for grayscale area scan cameras in Camera Link medium configuration mode. | Version 1.2 |
|  | CameraGrayLine | Camera image input interface for grayscale line scan GigE Vision cameras. | Version 1.2 SR5 |
|  | CameraGrayLineBase | Camera image input interface for grayscale line scan cameras in Camera Link base configuration mode. | Version 1.2 |
|  | CameraGrayLineFull | Camera image input interface for grayscale line scan cameras in Camera Link full configuration mode. | Version 1.2 |
|  | CameraGrayLineMedium | Camera image input interface for grayscale line scan cameras in Camera Link medium configuration mode. | Version 1.2 |
|  | CameraRgbArea | Camera image input interface for RGB area scan GigE Vision cameras. | Version 1.2 SR5 |
|  | CameraRgbAreaBase | Camera image input interface for RGB area scan cameras in Camera Link base configuration mode. | Version 1.2 |
|  | CameraRgbAreaMedium | Camera image input interface for RGB area scan cameras in Camera Link medium configuration mode. | Version 1.2 |
|  | CameraRgbLine | Camera image input interface for RGB line scan GigE Vision cameras. | Version 1.2 SR5 |
|  | CameraRgbLineBase | Camera image input interface for RGB line scan cameras in Camera Link base configuration mode. | Version 1.2 |
|  | CameraRgbLineMedium | Camera image input interface for RGB line scan cameras in Camera Link medium configuration mode. | Version 1.2 |
|  | CLHSDualCamera | Image data interface between CLHS camera and VisualApplets. | Version 3.0.3 |
|  | CLHSPulseIn | The operator manages the sending of pulse messages. | Version 3.0.3 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | CLHSPulseOut | The operator manages the receiving of pulse messages. | Version 3.0.3 |
|  | CLHSSingleCamera | Image data interface between CLHS camera and VisualApplets. | Version 3.0.3 |
|  | CxpCamera | This operator represents the image data interface between a CXP camera and VisualApplets. | Version 3.3 |
|  | CxpCameraMultiTap | This operator represents the image data interface between a CXP dual-tap camera and VisualApplets. | Version 4.0 |
|  | CxpAcquisitionStatus | Provides a signal that indicates whether an acquisition for a selected CXP camera port is started by the runtime environment. | Version 3.3 |
|  | CxpPortStatus | Monitors the status of a CXP channel. | Version 3.3 |
|  | CxpRxTrigger | Provides a receiver for trigger data from a CXP channel. | Version 3.3 |
|  | CxpTxTrigger | This operator provides a sender for trigger packets to a CXP channel. | Version 3.3 |
|  | CXPDualCamera | Image data interface between CXP dual channel camera and VisualApplets. | Version 2.1 |
|  | CXPQuadCamera | Image data interface between CXP camera(s) and VisualApplets. | Version 2.1 |
|  | CXPSingleCamera | Image data interface between CXP camera(s) and VisualApplets. | Version 2.1 |
|  | DigIOPort | Shows and sets the status of the TriggerIOs using parameters. | Version 1.2 |
|  | DmaFromPC | Image data input interface for transfer from host PC. | Version 1.2 |
|  | DmaToPC | Image data output interface for transfer to host PC. | Version 1.2 |
|  | GPI | Provides an interface to the digital inputs of microEnable or LightBridge. | Version 1.3 |
|  | GPO | Provides an interface to the digital outputs of microEnable or LightBridge. | Version 1.3 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | LED | Provides interface for accessing the 4 board LEDs via applet | Version 2.1 |
|  | NativeTrgPortIn | Provides an interface to the microEnable's digital inputs. | Version 1.2 |
|  | NativeTrgPortInExt | Provides an interface to the microEnable's digital inputs. | Version 1.2 |
|  | NativeTrgPortOut | Provides an interface to the microEnable's digital outputs and CC outputs. | Version 1.2 |
|  | RxLink | Provides a data link interface between a pixelPlant and frame grabber board. | Version 1.3 |
|  | TrgPortArea | Generates the trigger (Exsync) for the Camera. | Version 1.2 |
|  | TrgPortLine | Generates the trigger (Exsync) for the camera and is also responsible for assembling the acquired lines to images. | Version 1.2 |
|  | TriggerIn | Provides an interface to the microEnable's digital inputs. | Version 1.3 |
|  | TriggerOut | Provides an interface to the microEnable's digital outputs. | Version 1.3 |
|  | TxLink | Provides a data link interface between a pixelPlant and frame grabber board. | Version 1.3 |
|  | SignalToEvent | Generates software events for its input links. | Version 1.2 |

Table 29.1. Operators of Library Hardware Platform

# 29.1. Operator AppletProperties

**Operator Library: Hardware Platform**

The operator AppletProperties has no inputs and no outputs. It controls the applet and the board hardware setup.

The operator also provides a set of parameters that describe the applet (by partly user defined, partly automatically generated values).

This operator is mandatory in every design and is automatically instantiated by VisualApplets. The operator cannot be deleted.

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |
| mE5 marathon VCLx |
| mE5 marathon VCL |
| mE5 marathon VCX-QP |
| mE5 marathon VF2 |
| LightBridge VCL |
| mE5 ironman VQ8-CXP6D |
| mE5 ironman VQ8-CXP6B |
| mE5 ironman VD8-PoCL |



**Available Parameters**

Operator AppletProperties provides a specific set of parameters for every hardware platform.

The parameter descriptions below state for which hardware platforms a specific parameter is available.

## 29.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no Inputs or Outputs) |

## 29.1.2. Supported Link Format

None

## 29.1.3. Parameters

| VisualAppletsVersion | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |
| Version number of used VisualApplets version. | |

| VisualAppletsVersion |
|---|

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| ProjectName | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |

Name of the project as specified in menu *Design*, menu item *Properties*.

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| AppletVersion | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |

Version as specified in menu *Design*, menu item *Properties*.

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| AppletVendor | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |

Directly in the *Module Properties* dialog of this operator, you can enter information on the Applet vendor.

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| AppletAuthor | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |

Directly in the *Module Properties* dialog of this operator, you can enter information on the Applet author.

| AppletAuthor |
|---|

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| DesignClock (imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms) | |
|---|---|
| **Type** | static read parameter |
| **Default** | 312.5 |
| **Range** | [312.5; 400.0] |

The parameter represents the design system clock frequency in Mega Hertz. You can alter the design system clock frequency in VisualApplets, menu *Design*, menu item *Change FPGA Clock*. Using higher frequency settings as 135 MHz may result in a timing violation. In this case, reduce frequency.

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| DesignClock (mE5 platforms and LightBridge VCL) | |
|---|---|
| **Type** | static read parameter |
| **Default** | 125 |
| **Range** | [125;312.5] |

The parameter represents the design system clock frequency in Mega Hertz. You can alter the design system clock frequency in VisualApplets, menu *Design*, menu item *Change FPGA Clock*. Using higher frequency settings as 135 MHz may result in a timing violation. In this case, reduce frequency.

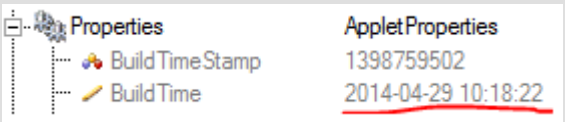
**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL

| BuildTime | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |

The BuildTime parameter is an automatically generated string showing when the applet has been build (date and time of latest net list generation).

The parameter value is only visible in the Framegrabber SDK. The following image shows the formatting of the string:



| BuildTime |
|---|

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| AppletUid | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |

Unique ID of the applet. A new ID is created with every new build (if the precondition check is set).

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| PcieInterfaceType (ironman platforms) | |
|---|---|
| **Type** | static write parameter |
| **Default** | Generation_2 |
| **Range** | {Generation_1, Generation_2} |

The parameter PcieInterfaceType allows you to select the PCIe interface type and thus the supported DMA bandwidth.

Generation_2 (default) stands for the PCIe Generation 2 protocol and provides a bandwidth of 3,600,000 bytes/s.

Generation_1 stands for PCIe Generation 1 protocol and provides a bandwidth of 1,800,000 bytes/s.

Generation_1 designs utilize slightly less FPGA resources. It is much easier to achieve timing closure for the applet with a Generation_1 design. Generation_2 provides maximal performance, but makes it more challenging to achieve timing closure for the applet. Multiple place and route synthesis runs will probably be necessary to meet the timing.

**Availability**

mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| PcieInterfaceType (marathon and LightBridge platforms) | |
|---|---|
| **Type** | static write parameter |
| **Default** | Generation_2 |
| **Range** | {Generation_1, Generation_2} |

The parameter PcieInterfaceType allows you to select the PCIe interface type and thus the supported DMA bandwidth.

Generation_1: The applet only supports PCIE generation 1. Thus, the maximum DMA transmission bandwidth is ca. 900 MByte/s (1MByte = 1^10Byte). The DmaToPC operator is limited so that LinkParallelism x LinkPixelWidth needs to be <= 64 bit.

Generation_2: The applet supports PCIE generation 2. Thus, the maximum DMA transmission bandwidth is ca. 1800 MByte/s (1MByte = 1^10Byte). The DmaToPC operator is limited so that LinkParallelism x LinkPixelWidth needs to be <= 128 bit.

| PcieInterfaceType (marathon and LightBridge platforms) |
|---|
| The implementation of Generation_1 consumes less resources than the implementation of Generation_2. The Generation_1 mode is helpful in designs that do not need the high bandwidth of Generation_2, but are short of resources. |



**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, and LightBridge VCL.

| ExtensionGpioType | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | OpenDrain |
| **Range** | {OpenDrain, PushPull} |

This parameter allows the user to select the circuitry type for the external extension GPIO connector. In PushPull mode, the GPIOs are driven directly.



**Parameter Only Affects the GPIO Connector**

This parameter has no effect on Front GPIO connector. Only the GPIO connector is influenced by this parameter.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| GpioType | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | OpenDrain |
| **Range** | {OpenDrain, PushPull} |

This parameter allows the user to select the circuitry type for the external GPIO connector. In PushPull mode, the GPIOs are driven directly.



**Parameter Only Affects the GPIO Connector**

This parameter has no effect on Front GPIO connector. Only the GPIO connector is influenced by this parameter.



**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| BuildTimeStamp | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | |



**Visibility**

This parameter is not displayed in the operator's parameter list.

| BuildTimeStamp |
|---|
| However, the parameter is visible in the runtime environment. |

BuildTimeStamp is an automatically generated identification number. It's a 32-bit time stamp that can be read out of the applet during runtime for checking the consistency between the loaded applet and the loaded applet runtime environment.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| ProjectNameHash | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |



**Visibility**

This parameter is not displayed in the operator's parameter list.

However, the parameter is visible in the runtime environment.

32-bit hash over the entry in ProjectName (menu Design, menu item Properties). This value can be read out of the applet during runtime for checking the consistency between the loaded applet and the loaded applet runtime environment.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| AppletUidHigh | |
|---|---|
| **Type** | static read parameter |
| **Default** | |
| **Range** | |



**Visibility**

This parameter is not displayed in the operator's parameter list.

However, the parameter is visible in the runtime environment.

Parameter AppletUid represents a 128-bit number. The upper 64 bit of this number are given out as a number in AppletUidHigh.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| AppletUidLow | |
|---|---|
| **Type** | static read parameter |
| **Default** | 1 |
| **Range** | {0; 2^32 – 1} |

**Visibility**

This parameter is not displayed in the operator's parameter list.

However, the parameter is visible in the runtime environment.

Parameter AppletUid represents a 128-bit number. The lower 64 bit of this number are given out as a number in AppletUidLow.

**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| FrontGpioPullControl | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | PullUp |
| **Range** | {PullUp, PullDown} |

- PullUp: On all Front GPIO inputs, pullup resistors are enabled to SLTRG_VIN (external 5–24 V input reference voltage).
- PullDown: On all front GPIO inputs, pulldown resistors are enabled to SLTRG_GND (isolated GND).

**Availability**

imaFlex CXP-12 Penta

| FrontGpioType | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | SingleEnded |
| **Range** | {SingleEnded, Differential} |

- Differential: All Front GPIO inputs are differential.
- SingleEnded: All Front GPIO inputs are single-ended (referenced to SLTRG_GND).

**Availability**

imaFlex CXP-12 Penta

| FrontGpioInversion | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Off |
| **Range** | {On, Off} |

- On: All Front GPIO Outputs are inverted polarity compared to FPGA output.
- Off: All Front GPIO Outputs are the same polarity compared to FPGA output.

| FrontGpioInversion |
| --- |
| **Availability**<br><br>imaFlex CXP-12 Penta |

### 29.1.4. Examples of Use

The use of operator AppletProperties is shown in the following examples:

- Section 10.4.1, 'CoaXPress Area Scan Cameras'

  Tutorial - Basic Acquisition
- Section 10.4.2, 'CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition
- Section 10.5.1, 'CoaXPress Area Scan Cameras'

  Tutorial - Basic Acquisition
- Section 10.5.2, 'CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition

# 29.2. Operator BoardStatus

**Operator Library: Hardware Platform**

The operator BoardStatus has no inputs and no outputs. It provides read-only parameters to monitor the current board status during runtime.

This operator is mandatory in every mE5 design and is automatically instantiated by VisualApplets. The operator cannot be deleted.

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |
| mE5 marathon VCL |
| mE5 marathon VCLx |
| mE5 marathon VCX-QP |
| mE5 marathon VF2 |
| LightBridge 2 VCL |
| mE5 ironman VQ8-CXP6D |
| mE5 ironman VQ8-CXP6B |
| mE5 ironman VD8-PoCL |



**Available Parameters**

Operator BoardStatus provides a specific set of parameters for every hardware platform.

The parameter descriptions below state for which hardware platforms a specific parameter is available.

## 29.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are no Inputs or Outputs) |

## 29.2.2. Supported Link Format

None

## 29.2.3. Parameters

| FpgaDNA | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | [0; 2^57-1] |

The parameter FpgaDNA provides the 57-bit unique FPGA DNA as an integer value.



**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| FpgaDNAHigh | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | [0; 2^32-1] |

This parameter FpgaDNAHigh provides the 32-bit MSB part of the 96-bit unique FPGA DNA as an integer value.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| FpgaDNALow | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | [0; 2^64-1] |

This parameter provides the 64-bit LSB part of the 96-bit unique FPGA DNA as an integer value.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| FpgaTemperature | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | imaFlex CXP-12 Quad and imaFlex CXP-12 Penta: [0; 200];<br><br>mE5 marathon, mE5 ironman and LightBridge VCL: [0; 512] |

The parameter FpgaTemperature provides the current FPGA temperature in degrees Celsius (°C).



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| FpgaVccInt | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | [0; 5] |

The parameter FpgaVccInt provides the current VCC INT voltage of the FPGA in Volt.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| FpgaVccAux | |
|---|---|
| **Type** | dynamic read parameter |

| FpgaVccAux | |
|---|---|
| **Default** | |
| **Range** | [0; 5] |

The parameter FpgaVccAux provides the current VCC AUX voltage of the FPGA in Volt.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL, mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B

| FpgaVccBram | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | [0; 5] |

The parameter provides the current VCC BRAM voltage of the FPGA in Volt.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL

| ExternalPowerGood | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {GOOD, NO_POWER} |

- GOOD: External power connector is plugged on the board and provides 12 V power.
- NO_POWER: Board does not detect an external 12-V-power-connector. Either the board is not plugged in or it is depowered.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| ExtensionGpioBoardPresent | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {YES, NO} |

- YES: Extension GPIO board is detected to be plugged in on the imaFlex CXP-12 Quad / imaFlex CXP-12 Penta board.
- NO: extension GPIO board is not plugged in and is not detected.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| PcieNegotiatedLinkWidth | |
|---|---|
| **Type** | dynamic read parameter |

| PcieNegotiatedLinkWidth | |
|---|---|
| **Default** | 0 |
| **Range** | {0, 1, 2, 4, 8} |

This parameter provides the amount of PCIe lanes actively used by the frame grabber and the host PC system.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| PcieNegotiatedLinkSpeed | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 8 |
| **Range** | {2.5, 5, 8} |

This parameter provides the PCIe speed in Gigabit per second.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| PcieTrainedPayloadSize | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 128 |
| **Range** | imaFlex CXP-12 Quad and imaFlex CXP-12 Penta: {128, 256, 512, 1024}; mE5 marathon, mE5 ironman, and LightBridge VCL: {128, 256, 512} |

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



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta, mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL only

| PcieTrainedRequestSize | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 128 |
| **Range** | {128, 256, 512, 1024, 2048, 4096} |

This parameter provides the size of the PCIe packet requests [in bytes] used for the data transmission between the frame grabber and the PCIe bridge.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| BoardPower | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 100} |

The parameter BoardPower measures the board total power consumption in Watt.

| BoardPower |
|---|
| **Availability**<br><br>mE5 ironman VD8-PoCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B |

| CxpChipTemperature | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 512} |

The parameter CxpChipTemperature measures the temperature near the CXP front-end chips in degrees Celsius (°C).

**Availability**

mE5 ironman VQ8-CXP6D and mE5 ironman VQ8-CXP6B only

| RamChipTemperature | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 512} |

The parameter RamChipTemperature measures the temperature near the RAM chips in degrees Celsius (°C).

**Availability**

mE5 ironman VQ8-CXP6D and mE5 ironman VQ8-CXP6B only

| CxpPowerRegulatorTemperature | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 512} |

The parameter CxpPowerRegulatorTemperature measures the temperature near the CXP power chips in degrees Celsius (°C).

**Availability**

mE5 ironman VQ8-CXP6D and mE5 ironman VQ8-CXP6B only

| PowerRegulatorTemperature | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 512} |

The parameter PowerRegulatorTemperature measures the temperature near the board power regulator chips in degrees Celsius (°C).

**Availability**

mE5 ironman VQ8-CXP6D and mE5 ironman VQ8-CXP6B only

| ChannelCurrent | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 30} |

The parameter ChannelCurrent measures the CXP power chip current in Ampere of each channel available on the board. (4 channels are present on all SiSo CXP grabbers.)



**Availability**

mE5 ironman VQ8-CXP6D and mE5 ironman VQ8-CXP6B only

| ChannelVoltage | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 30} |

The parameter ChannelVoltage measures the CXP power chip voltage in Volt of each channel available on the board. (4 channels are present on all SiSo CXP grabbers.)



**Availability**

mE5 ironman VQ8-CXP6D and mE5 ironman VQ8-CXP6B only

| PcieCurrentLinkWidth | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 4] |

The parameter provides the amount of PCIe lanes used by the framegrabber.



**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL

| PcieCurrentLinkSpeed | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 5 |
| **Range** | [2.5; 5] |

PCIe speed in Gigabit per second.



**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL

| PcieLinkGen2Capable | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | YES |
| **Range** | {YES, NO} |

Capability of the framegrabber to support PCIe generation 2 link.

| PcieLinkGen2Capable |
|---|

**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL

| PcieLinkPartnerGen2Capable | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | YES |
| **Range** | {YES, NO} |

Capability of the bridge (device connected to the framegrabber through PCIe link) to support PCIe generation 2 link.

**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL

| AlternativeBoardDetection | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | OFF |
| **Range** | {OFF, ON} |

Parameter *AlternativeBoardDetection* informs which board detection algorithm is activated for board detection.

- OFF: Standard board detection algorithm is activated (default).
- ON: Alternative board detection algorithm is activated.

**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL only

| ExtensionConnectorPresent | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | YES |
| **Range** | {YES, NO} |

The parameter indicates the existence of the extension GPIO board connected to GPIO slot connector (labeled as GPIO in the board documentation).

**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge 2 VCL only

| PoCLStatePortA | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | Initialize |
| **Range** | {PoCL_Disabled, Initialize, PoCL_Connection_Sense, PoCL_Wait_for_Connection, PoCL_Camera_Detected, PoCL_Camera_Clock_Detected, CL_Wait_for_Connection, CL_Camera_Detected, CL_Camera_Clock_Detected} |

**PoCLStatePortA**

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



**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCLx, LightBridge 2 VCL



**Using Power over Camera Link (PoCL)**

In applets built with VisualApplets version 3.0.6 or higher, PoCL is disabled per default. Thus, read parameter **PoCLStatePortA** displays value **PoCL_Disabled** per default. If the user of the applet wants to use the applet with PoCL support, he/she needs to enable PoCL support via microDiagnostics. This is possible as of Framegrabber SDK version 5.5.1 (or higher) (with version 5.10, the product name has been changed from **Basler Runtime** to **Basler Framegrabber SDK**).

Thus, if the applet user is going to use PoCL:

- If the applet user has Framegrabber SDK version 5.5.1 or higher installed on his host PC, you may use VisualApplets 3.0.6 or higher to design and build the applet.
- If the applet user is definitely not going to update the Framegrabber SDK version on his host PC to 5.5.1 or higher, but wants to use PoCL, you need to design the applet with a VisualApplets version 3.0.4 (or prior).

It will not be possible to use applets build with VisualApplets 3.0.6 or higher in PoCL mode together with a Framegrabber SDK version lower than 5.5.1.

**PoCL_Disabled**: If PoCL is not enabled for the board, this state will be displayed during runtime.



**Enabling PoCL Support**

Starting from Framegrabber SDK version 5.5.1, PoCL support is disabled per default for all mE5 marathon and LightBridge CL frame grabbers. To enable PoCL support, hardware users using these frame grabber models need to enable PoCL support in microDiagnostics, menu **Tools -> Board Settings: Enable PoCL Detection** when using Framegrabber SDK version 5.5.1 or higher.

If PoCL has been enabled in microDiagnostics (or the applet has been built with VisualApplets 3.0.4 or prior), the individual states of the Power over Camera Link (PoCL) state machine indicate the following:

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

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

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

(The controller applies a test current and waits for 600 ms. Then, the voltage drop is measured. There are two thresholds: Is the measured value between both thresholds, the PoCL state machine switches to state PoCL_Wait_for_Connection. Is the measured value lower than both thresholds, the

**PoCLStatePortA**

PoCL state machine switches to state CL_Wait_for_Connection. Is the measured value higher than both thresholds, the connection is either broken or disturbed. In this case, the PoCL state machine switches to state Initialize and starts again.)

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

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

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

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

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

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

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

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

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

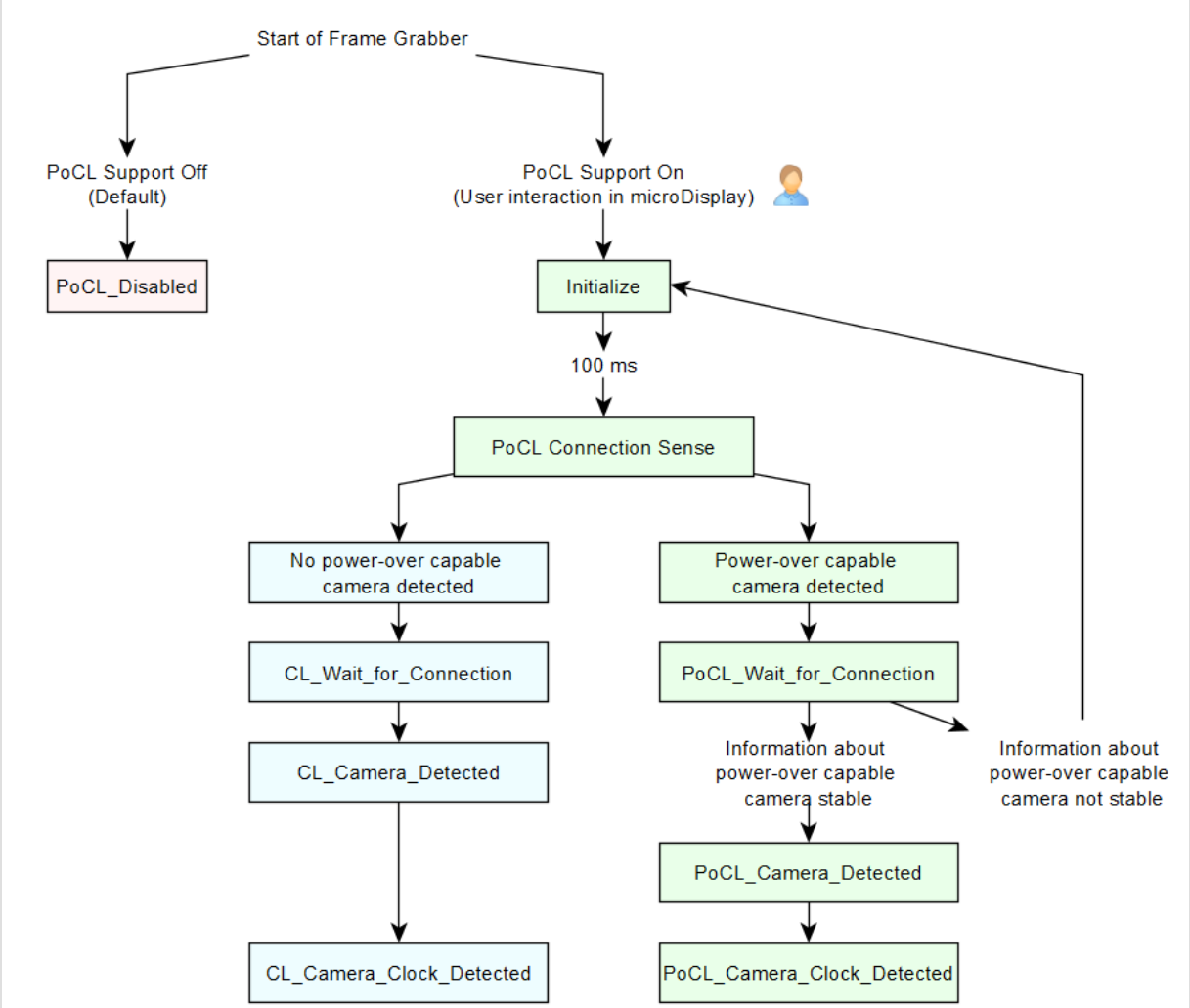
- If a clock is detected (within maximally 4s), the camera is ready for operation. The state machine switches to state CL_Camera_Clock_Detected.
- If during 4s no clock is detected, the camera is not ready for operation. The state machine switches to state Initialize and starts again.

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

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

If PoCL support is enabled, the PoCL state machine decision flow runs as follows:

**PoCLStatePortA**




### Power Watchdog

Additional security mechanism: The voltage level is permanently monitored.

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

| **PoCLStatePortB** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | Initialize |
| **Range** | {PoCL_Disabled, Initialize, PoCL_Connection_Sense, PoCL_Wait_for_Connection, PoCL_Camera_Detected, PoCL_Camera_Clock_Detected, CL_Wait_for_Connection, CL_Camera_Detected, CL_Camera_Clock_Detected} |

**PoCLStatePortB**

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



**Availability**

mE5 marathon VCL, mE5 marathon VCLx, LightBridge 2 VCL



**Using Power over Camera Link (PoCL)**

In applets built with VisualApplets version 3.0.6 or higher, PoCL is disabled per default. Thus, read parameter **PoCLStatePortB** displays value **PoCL_Disabled** per default. If the user of the applet wants to use the applet with PoCL support, he/she needs to enable PoCL support via microDiagnostics. This is possible as of Framegrabber SDK version 5.5.1 (or higher) (with version 5.10, the product name has been changed from **Basler Runtime** to **Basler Framegrabber SDK**).

Thus, if the applet user is going to use PoCL:

- If the applet user has Framegrabber SDK version 5.5.1 or higher installed on his host PC, you may use VisualApplets 3.0.6 or higher to design and build the applet.
- If the applet user is definitely not going to update the Framegrabber SDK version on his host PC to 5.5.1 or higher, but wants to use PoCL, you need to design the applet with a VisualApplets version 3.0.4 (or prior).

It will not be possible to use applets build with VisualApplets 3.0.6 or higher in PoCL mode together with a Framegrabber SDK version lower than 5.5.1.

**PoCL_Disabled**: If PoCL is not enabled for the board, this state will be displayed during runtime.



**Enabling PoCL Support**

Starting from Framegrabber SDK version 5.5.1, PoCL support is disabled per default for all mE5 marathon and LightBridge CL frame grabbers. To enable PoCL support, hardware users using these frame grabber models need to enable PoCL support in microDiagnostics, menu **Tools -> Board Settings: Enable PoCL Detection** when using Framegrabber SDK version 5.5.1 or higher.

If PoCL has been enabled in microDiagnostics (or the applet has been built with VisualApplets 3.0.4 or prior), the individual states of the Power over Camera Link (PoCL) state machine indicate the following:

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

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

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

(The controller applies a test current and waits for 600 ms. Then, the voltage drop is measured. There are two thresholds: Is the measured value between both thresholds, the PoCL state machine switches to state PoCL_Wait_for_Connection. Is the measured value lower than both thresholds, the

**PoCLStatePortB**

PoCL state machine switches to state CL_Wait_for_Connection. Is the measured value higher than both thresholds, the connection is either broken or disturbed. In this case, the PoCL state machine switches to state Initialize and starts again.)

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

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

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

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

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

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

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

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

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

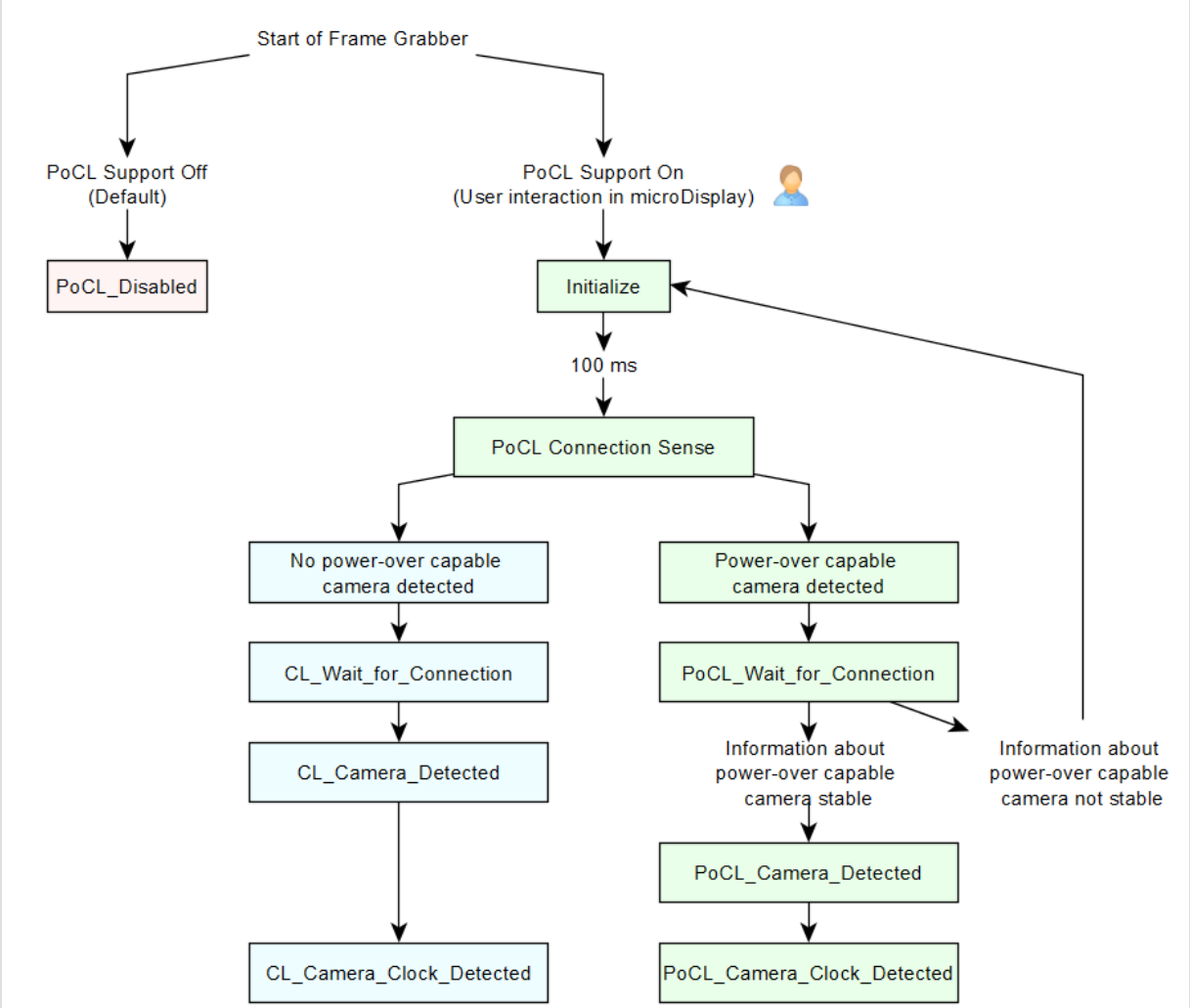
- If a clock is detected (within maximally 4s), the camera is ready for operation. The state machine switches to state CL_Camera_Clock_Detected.
- If during 4s no clock is detected, the camera is not ready for operation. The state machine switches to state Initialize and starts again.

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

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

If PoCL support is enabled, the PoCL state machine decision flow runs as follows:

**PoCLStatePortB**




### Power Watchdog

Additional security mechanism: The voltage level is permanently monitored.

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

| **PoCXPStatePort_0** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | BOOTING |
| **Range** | {BOOTING, NOCABLE, NOPOCXP, POCXPOK, MIN_CURR, MAX_CURR, LOW_VOLT, OVER_VOLT, ADC_Chip_Error} |
| The parameter represents the current state of the power over CoaXPress state machine on the frame grabber's port[0] connector. | |

| PoCXPStatePort_0 |
|---|
| **Availability**<br><br>mE5 marathon VCX-QP |

| PoCXPCurrentPort_0 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [-1000.0 mA; +1000.0 mA] |
| The parameter measures the currently flowing current on port[0] (in mA). | |
| **Availability**<br><br>mE5 marathon VCX-QP | |

| PoCXPVoltagePort_0 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 24 V |
| **Range** | [0.0 V; 30.0 V] |
| The parameter measures the voltage of the CXP power regulator of port[0] (in V). When the port is not connected, the measured voltage will be 24 V. | |
| **Availability**<br><br>mE5 marathon VCX-QP | |

| PoCXPStatePort_1 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | BOOTING |
| **Range** | {BOOTING, NOCABLE, NOPOCXP, POCXPOK, MIN_CURR, MAX_CURR, LOW_VOLT, OVER_VOLT, ADC_Chip_Error} |
| The parameter represents the current state of the power over CoaXPress state machine on the frame grabber's port[1] connector. | |
| **Availability**<br><br>mE5 marathon VCX-QP | |

| PoCXPCurrentPort_1 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [-1000.0 mA; +1000.0 mA] |
| The parameter measures the currently flowing current on port[1] (in mA). | |
| **Availability**<br><br>mE5 marathon VCX-QP | |

| PoCXPVoltagePort_1 | |
|---|---|
| **Type** | dynamic read parameter |

| PoCXPVoltagePort_1 | |
|---|---|
| **Default** | 24 V |
| **Range** | [0.0 V; 30.0 V] |

The parameter measures the voltage of the CXP power regulator of port[1] (in V). When the port is not connected, the measured voltage will be 24 V.



**Availability**

mE5 marathon VCX-QP

| PoCXPStatePort_2 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | BOOTING |
| **Range** | {BOOTING, NOCABLE, NOPOCXP, POCXPOK, MIN_CURR, MAX_CURR, LOW_VOLT, OVER_VOLT, ADC_Chip_Error} |

The parameter represents the current state of the power over CoaXPress state machine on the frame grabber's port[2] connector.



**Availability**

mE5 marathon VCX-QP

| PoCXPCurrentPort_2 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [-1000.0 mA; +1000.0 mA] |

The parameter measures the currently flowing current on port[2] (in mA).



**Availability**

mE5 marathon VCX-QP

| PoCXPVoltagePort_2 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 24 V |
| **Range** | [0.0 V; 30.0 V] |

The parameter measures the voltage of the CXP power regulator of port[2] (in V). When the port is not connected, the measured voltage will be 24 V.



**Availability**

mE5 marathon VCX-QP

| PoCXPStatePort_3 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | BOOTING |
| **Range** | {BOOTING, NOCABLE, NOPOCXP, POCXPOK, MIN_CURR, MAX_CURR, LOW_VOLT, OVER_VOLT, ADC_Chip_Error} |

The parameter represents the current state of the power over CoaXPress state machine on the frame grabber's port[3] connector.

| PoCXPStatePort_3 |
|---|
| **Availability**<br><br>mE5 marathon VCX-QP |

| PoCXPCurrentPort_3 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [-1000.0 mA; +1000.0 mA] |
| The parameter measures the currently flowing current on port[3] (in mA). | |
| **Availability**<br><br>mE5 marathon VCX-QP | |

| PoCXPVoltagePort_3 | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 24 V |
| **Range** | [0.0 V; 30.0 V] |
| The parameter measures the voltage of the CXP power regulator of port[3] (in V). When the port is not connected, the measured voltage will be 24V. | |
| **Availability**<br><br>mE5 marathon VCX-QP | |

### 29.2.4. Examples of Use

The use of operator BoardStatus is shown in the following examples:

- Section 10.4.1, 'CoaXPress Area Scan Cameras'

  Tutorial - Basic Acquisition
- Section 10.4.2, 'CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition
- Section 10.5.1, 'CoaXPress Area Scan Cameras'

  Tutorial - Basic Acquisition
- Section 10.5.2, 'CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition

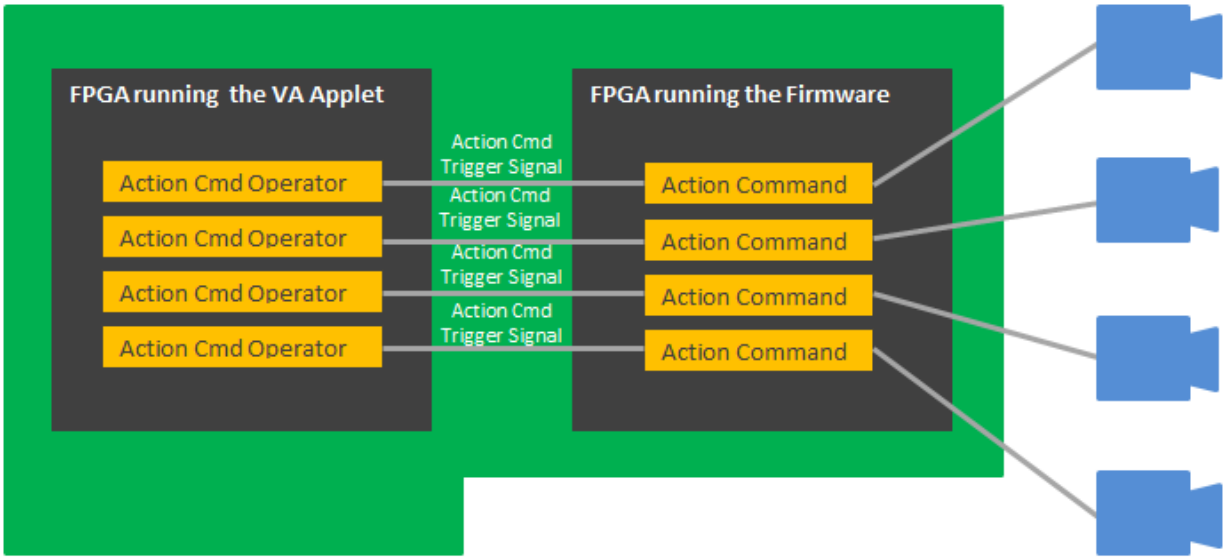
## 29.3. Operator ActionCommand

**Operator Library: Hardware Platform**

The operator *Action Command* sends an action command trigger signal to the firmware. On receiving this trigger signal, the firmware sends a preconfigured action command to the GigE Vision camera.

The operator fires a trigger at detecting a rising edge at its input port. The operator guarantees that the trigger signal has a minimum pulse width of three clock cycles. This ensures the firmware detects the trigger signal.

For each camera channel, one action command is available. Maximally four resources of type Action Command are available. Maximally four *Action Command* operators can be used in a design. The index number of the action command is the same as the index number of the connected camera port.




For each action command the camera receives, it returns an action command acknowledge packet.

The operator offers a read parameter *PendingActionCmdCount* that displays the status of counter *PendingActionCmdCount*: As soon as operator *Action Command* sends a trigger signal to the firmware, counter *PendingActionCmdCount* is incremented. As soon as the receiving camera sends the acknowledge packet, counter *PendingActionCmdCount* is decremented. The counter is protected against overflow and underflow: At reaching the maximal value, the maximal value is hold. When the counter has already value zero, the counter doesn't decrement further.

Operator *Action Command* is a mere trigger operator. The content (function) of the action command is defined by software.

The firmware sends the action command at the rising edge of the action command trigger signal it receives from the *Action Command* operator.

The firmware sends the action command to the camera with a fix latency of 2000 ns +/- 16 ns. This means, the time gap between the detection of the rising edge of the action command trigger signal by the firmware and the actual sending of the action command by the firmware is always 2000 ns +/- 16 ns.

**Keep software packets on transmitter channel small during acquisition**

The transmitter channel between firmware and camera is used for sending software packets as well as action command packets. **To ensure the fix latency can be kept, make sure the software packets do not exceed a payload size of approximately 100 Byte during acquisition.**



**Interval of minimum 2000 ns between trigger signals required**

Make sure between the sending of two action command trigger signals (back-to-back trigger signals) by the operator *Action Command*, there is a minimum time gap of 2000 ns.

| Available for Hardware Platforms |
|---|
| microEnable IV VQ4-GE |
| microEnable IV VQ4-GPoE |

### 29.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | Signal Input Link I, Signal Input |

### 29.3.2. Supported Link Format

| Link Parameter | Input Link Signal Input Link I |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.3.3. Parameters

| PendingActionCmdCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | [0; 65535] |

Parameter *PendingActionCmdCount* displays the status of counter *PendingActionCmdCount*:
As soon as operator *Action Command* sends a trigger signal to the firmware, counter *PendingActionCmdCount* is incremented. As soon as the receiving camera sends the acknowledge packet, counter *PendingActionCmdCount* is decremented.

The counter is protected against overflow and underflow: At reaching the maximal value, the maximal value is hold. When the counter has already value zero, the counter doesn't decrement further.

### 29.3.4. Examples of Use

The use of operator ActionCommand is shown in the following examples:

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

# 29.4. Operator CameraControl

**Operator Library: Hardware Platform**

The CameraControl operator provides the camera control [CC] interface to the connected camera. When using multiple instances of the CameraControl operator, the assignment to the proper camera port can be made in the resource dialog window. Thus, this operator requires one VisualApplets resource of type **`CameraControl`**. The resource index 0 maps to camera port A and the resource index 1 maps to camera port B. Check Section 3.8, 'Allocation of Device Resources' for more information.

| Available for Hardware Platforms |
|---|
| mE5 marathon VCLx |
| mE5 marathon VCL |
| LightBridge VCL |
| microEnable 5 ironman VD8-PoCL |
| microEnable IV VD4-CL/-PoCL |
| microEnable IV VD1-CL |



**One Instance with microEnable 5 VD8-PoCL**

Due to the board design of microEnable 5 VD8-PoCL, only one instance of this operator is available on this platform.

## 29.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | CC[1-4], signal input |

| Port Name | Direction | Type | Description |
|---|---|---|---|
| CC1 | Input | OM | Camera Link CC1 channel. |
| CC2 | Input | OM | Camera Link CC2 channel. |
| CC3 | Input | OM | Camera Link CC3 channel. |
| CC4 | Input | OM | Camera Link CC4 channel. |

**Synchronous and Asynchronous Inputs:**

All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized. See also Section 3.6.4, 'M-type Operators with Multiple Inputs'.

## 29.4.2. Supported Link Format

| Link Parameter | Input Link CC[1-4] |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |

| Link Parameter | Input Link CC[1-4] |
|---|---|
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.4.3. Parameters

None

### 29.4.4. Examples of Use

The use of operator CameraControl is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 29.5. Operator BaseGrayCamera

**Operator Library: Hardware Platform**

The operator transfers image data from a Camera Link BASE GRAY configuration camera into VisualApplets. Any BASE GRAY configurations of the Camera Link standard are supported.

The operator uses one ressource of type CameraPort exclusively. You can select Camera Link port A ( 0 ) or Camera Link port B ( 1 ) of the frame grabber. The same ressource index can only be used once per applet.

microEnable 5 ironman VD8-PoCL: On mE5 VD8-PoCL (ironman), only port A offers all camera functions, i.e., CC signals and serial interface. On mE5 VD8-PoCL (ironman), port B offers neither CC signals nor a serial interface. Nevertheless, the camera can be operated in free-run mode when the camera has been pre-programmed before. This limitations of port A are only true for mE5 VD8-PoCL (ironman), not for any other frame grabber models (LightBridge, marathon ...) using the operator.

| Available for Hardware Platforms |
|---|
| mE5 marathon VCLx |
| mE5 marathon VCL |
| LightBridge VCL |
| mE5 VD8-PoCL (ironman) |

## 29.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.5.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | Any❶ |
| **Arithmetic** | unsigned |
| **Parallelism** | Any, see parameter description |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D,VALT_LINE1D} |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | Any |
| **Max. Img Height** | Any |

❶ Parameter BitWidth can be set to a larger or smaller value than the actual bitwidth sent by the camera.

If the parameter is set to a smaller value than the actual bitwidth sent by the camera, the least significant bits (LSB) of the pixel are cut off. Only the most signicicant bits (MSB) are transferred into VisualApplets.

If the parameter is set to a bigger value than the actual bitwidth sent by the camera, the original pixel is set on the MSB position. The LSBs are filled with zeros.

In both cases, the relative brightness stays the same, i.e., white pixels remain withe and dark pixels remain dark.

### 29.5.3. Parameters

| DvalMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |
| The parameter specifies whether DVAL is used to mask valid pixels or if it is ignored. When DVAL is ignored, the valid pixels are selected according to LVAL and FVAL statuses. | |

| FvalMode | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | CameraLinkStandard |
| **Range** | {CameraLinkStandard, IgnoreFval} |

- CameraLinkStandard

  The operator expects frame valid (FVAL) signals from the camera. In this case, the operator will output frames according to the FVAL size.

- IgnoreFval

  If this setting is used, the operator will ignore any FVAL signals from the camera. All input lines will be used and merged into a sigle image with infinite height.



**Only available for Output Image Protocol VALT_IMAGE2D**

This parameter is only available if the output is set to image protocol VALT_IMAGE2D. If VALT_LINE1D is selected, the parameter is disabled and the FVAL is ignored anyway.



**VisualApplets Link Rules Violation**

When the camera operator outputs a frame of infinite height, the Max. Img Height defined in the link properties might be exceeded. This causes a violation of the VisualApplets link rules. Ensure to use valid heights. Use operator *SplitImage* to limit the image height and split the endless image into multiple chunks.

The intention of this parameter is to allow the usage of area scan an line scan cameras dynamically in the same applet.

| BaseMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Tap1x8bit |
| **Range** | {Tap1x8bit, Tap1x10bit, Tap1x12bit, Tap1x14bit, Tap1x16bit, Tap2x8bit, Tap2x10bit, Tap2x12bit, Tap3x8bit} |
| The parameter specifies the BASE configuration mode. | |

| CameraStatus | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | LightBridge VCL/mE5 marathon VCL/mE5 marathon VCLx: [0; 2^8-1]<br><br>mE5 VD8 ironman: [0;2^4-1] |
| bit[0]: PCLK [0] available<br><br>bit[1]: FVAL [0] current value | |

| CameraStatus |
| --- |
| bit[2]: LVAL [0] current value<br>bit[3]: DVAL [0] current value<br>bit[4..7]: reserved (LightBridge and marathon only) |

| CameraLinkCoreReset | |
| --- | --- |
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0; 1] |
| Reserved for support issues. | |

| PixelClock | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel in MHz. | |

| MinimalParallelism | |
| --- | --- |
| **Type** | static read parameter |
| **Default** | 3 |
| **Range** | [1; 1024] |
| Minimal parallelism for the output link O to be able to transport the maximal bandwidth of the camera without losing data. This value depends on the currently selected design clock frequency. The higher the frequency the lower the parallelism value can become. | |

### 29.5.4. Examples of Use

The use of operator BaseGrayCamera is shown in the following examples:

- Section 10.3.1.1, 'Grayscale Camera Link Base Area'

  Tutorial - Basic Acquisition

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

## 29.6. Operator BaseRgbCamera

**Operator Library: Hardware Platform**

The operator transfers image data from a Camera Link BASE RGB configuration camera into VisualApplets. All RGB BASE configurations of the Camera Link standard are supported.

The operator uses one resource of type CameraPort exclusively. You can select Camera Link port A ( 0 ) or Camera Link port B ( 1 ) of the frame grabber. The same resource index can only be used once per applet.

microEnable 5 ironman VD8-PoCL: On mE5 VD8-PoCL (ironman), only port A offers all camera functions, i.e., CC signals and serial interface. On mE5 VD8-PoCL (ironman), port B offers neither CC signals nor a serial interface. Nevertheless, the camera can be operated in free-run mode when the camera has been pre-programmed before. These limitations of port B are only true for mE5 VD8-PoCL (ironman), not for any other frame grabber models (LightBridge, marathon ...) using the operator.



**Definition of "Tap"**

The meaning of **one tap**, as used in this manual, is one decoded color component of any bit width.

Note that in the CameraLink Specification Version 2.1, **one tap** is defined as one color pixel with all color components included.

The following table shows the mapping between the VisualApplets notation and the CameraLink Specification Version 2.1:

| VA notation | CameraLink notation |
|---|---|
| Tap3x8 ..... | 8-bit RGB Medium 1 tap |

Table 29.2. Mapping VA notation and CL Specification Version 2.1

The following figure shows the pixel mapping in this mode (Tap3x8bit):




| Available for Hardware Platforms |
|---|
| mE5 marathon VCLx |
| mE5 marathon VCL |
| LightBridge VCL |
| mE5 VD8-PoCL (ironman) |

### 29.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

### 29.6.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | [3,63]❶ |
| **Arithmetic** | unsigned |
| **Parallelism** | Any, see parameter description |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D,VALT_LINE1D} |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | Any |
| **Max. Img Height** | Any |

❶ The bitwidth must be a multiple of 3 due to the RGB tuple.

Parameter BitWidth can be set to a larger or smaller value than the actual bitwidth sent by the camera.

If the parameter is set to a smaller value than the actual bitwidth sent by the camera, the least significant bits (LSB) of the pixel components are cut off. Only the most signicicant bits (MSB) per pixel component are transferred into VisualApplets.

If the parameter is set to a bigger value than the actual bitwidth sent by the camera, the original pixel components are set on the MSB position of the outgoing components. The LSBs are filled with zeros.

In both cases, the relative brightness stays the same, i.e., white pixels remain withe and dark pixels remain dark.

### 29.6.3. Parameters

| DvalMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |
| The parameter specifies whether DVAL is used to mask valid pixels or if it is ignored. When DVAL is ignored, the valid pixels are selected according to LVAL and FVAL statuses. | |

| FvalMode | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | CameraLinkStandard |
| **Range** | {CameraLinkStandard, IgnoreFval} |

- CameraLinkStandard

  The operator expects frame valid (FVAL) signals from the camera. In this case, the operator will output frames according to the FVAL size.

| FvalMode |
|---|
| • IgnoreFval<br><br>If this setting is used, the operator will ignore any FVAL signals from the camera. All input lines will be used and merged into a sigle image with infinite height.<br><br><br>**Only available for Output Image Protocol VALT_IMAGE2D**<br><br>This parameter is only available if the output is set to image protocol VALT_IMAGE2D. If VALT_LINE1D is selected, the parameter is disabled and the FVAL is ignored anyway.<br><br><br>**VisualApplets Link Rules Violation**<br><br>When the camera operator outputs a frame of infinite height, the Max. Img Height defined in the link properties might be exceeded. This causes a violation of the VisualApplets link rules. Ensure to use valid heights. Use operator *SplitImage* to limit the image height and split the endless image into multiple chunks.<br><br>The intention of this parameter is to allow the usage of area scan an line scan cameras dynamically in the same applet. |

| BaseMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Tap3x8bit |
| **Range** | {Tap3x8bit} |
| The parameter specifies the BASE configuration mode. | |

| CameraStatus | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | LightBridge/marathon: [0;2^8-1]<br><br>mE5 VD8 ironman: [0;2^4-1] |
| bit[0]: PCLK available<br><br>bit[1]: FVAL current value<br><br>bit[2]: LVAL current value<br><br>bit[3]: DVAL current value<br><br>bit[4-7]: reserved (LightBridge and marathon only) | |

| CameraLinkCoreReset | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0; 1] |
| Reserved for support issues. | |

| PixelClock | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |

| PixelClock |
| --- |
| Pixel clock rate of camera channel in MHz. |

| MinimalParallelism | |
| --- | --- |
| **Type** | static read parameter |
| **Default** | 1 |
| **Range** | [1; 1024] |
| Minimal parallelism for the output link O to be able to transport the maximal bandwidth of the camera without losing data. This value depends on the currently selected design clock frequency. The higher the frequency the lower the parallelism value can become. | |

# 29.7. Operator MediumGrayCamera

**Operator Library: Hardware Platform**

The operator transfers image data from a Camera Link MEDIUM GRAY configuration camera into VisualApplets. Any MEDIUM GRAY configurations of the Camera Link standard are supported.

The operator uses 2 ressources of type CameraPort exclusively. Thus, both ports of the frame grabber, port A (0) and port B (1), are occupied and cannot be used by other operators. The same ressource index can only be used once per applet.

| Available for Hardware Platforms |
|---|
| mE5 marathon VCLx |
| mE5 marathon VCL |
| LightBridge VCL |
| mE5 VD8-PoCL (ironman) |

## 29.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.7.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | Any❶ |
| **Arithmetic** | unsigned |
| **Parallelism** | Any, see parameter description |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D,VALT_LINE1D} |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | Any |
| **Max. Img Height** | Any |

❶ Parameter BitWidth can be set to a larger or smaller value than the actual bitwidth sent by the camera.

If the parameter is set to a smaller value than the actual bitwidth sent by the camera, the least significant bits (LSB) of the pixel are cut off. Only the most signicicant bits (MSB) are transferred into VisualApplets.

If the parameter is set to a bigger value than the actual bitwidth sent by the camera, the original pixel is set on the MSB position. The LSBs are filled with zeros.

In both cases, the relative brightness stays the same, i.e., white pixels remain withe and dark pixels remain dark.

### 29.7.3. Parameters

| DvalMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |
| The parameter specifies whether DVAL is used to mask valid pixels or if it is ignored. When DVAL is ignored, the valid pixels are selected according to LVAL and FVAL statuses. | |

| FvalMode | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | CameraLinkStandard |
| **Range** | {CameraLinkStandard, IgnoreFval} |

- CameraLinkStandard

  The operator expects frame valid (FVAL) signals from the camera. In this case, the operator will output frames according to the FVAL size.

- IgnoreFval

  If this setting is used, the operator will ignore any FVAL signals from the camera. All input lines will be used and merged into a sigle image with infinite height.



**Only available for Output Image Protocol VALT_IMAGE2D**

This parameter is only available if the output is set to image protocol VALT_IMAGE2D. If VALT_LINE1D is selected, the parameter is disabled and the FVAL is ignored anyway.



**VisualApplets Link Rules Violation**

When the camera operator outputs a frame of infinite height, the Max. Img Height defined in the link properties might be exceeded. This causes a violation of the VisualApplets link rules. Ensure to use valid heights. Use operator *SplitImage* to limit the image height and split the endless image into multiple chunks.

The intention of this parameter is to allow the usage of area scan an line scan cameras dynamically in the same applet.

| MediumMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Tap4x8bit |
| **Range** | {Tap3x10bit, Tap3x12bit, Tap4x8bit,Tap4x10bit, Tap4x12bit} |
| The parameter specifies the MEDIUM configuration mode. | |

| CameraStatus | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | LightBridge/marathon: [0;2^24-1]<br><br>mE5 VD8 ironman: [0;2^8-1] |
| Parameter CameraStatus depends on the frame grabber hardware you are programming for. Find below the descriptions first for marathon and Lightbridge, than for mE5 VD8-PoCL (ironman).<br><br>**marathon and Lightbridge:** | |

| CameraStatus |
|---|
| bit[0]: PCLK [0] available |
| bit[1]: FVAL [0] current value |
| bit[2]: LVAL [0] current value |
| bit[3]: DVAL [0] current value |
| bit[4-11]: reserved |
| bit[12]: PCLK [1] available |
| bit[13]: FVAL [1] current value |
| bit[14]: LVAL [1] current value |
| bit[15]: DVAL [1] current value |
| bit[16-23]: reserved |
| **mE5 VD8-PoCL (ironman):** |
| bit[0]: PCLK [0] available |
| bit[1]: FVAL [0] current value |
| bit[2]: LVAL [0] current value |
| bit[3]: DVAL [0] current value |
| bit[4]: PCLK [1] available |
| bit[5]: FVAL [1] current value |
| bit[6]: LVAL [1] current value |
| bit[7]: DVAL [1] current value |

| CameraLinkCoreReset | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0; 1] |
| Reserved for support issues. | |

| PixelClockX | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel X in MHz. | |

| PixelClockY | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel Y in MHz. | |

| MinimalParallelism | |
|---|---|
| **Type** | static read parameter |

| MinimalParallelism | |
|---|---|
| **Default** | 3 |
| **Range** | [1; 1024] |
| Minimal parallelism for the output link O to be able to transport the maximal bandwidth of the camera without losing data. This value depends on the currently selected design clock frequency. The higher the frequency the lower the parallelism value can become. | |

### 29.7.4. Examples of Use

The use of operator MediumGrayCamera is shown in the following examples:

- Section 10.3.1.3, 'Grayscale Camera Link Medium Area'

  Tutorial - Basic Acquisition

## 29.8. Operator MediumRgbCamera

**Operator Library: Hardware Platform**

The operator transfers image data from a Camera Link MEDIUM RGB configuration camera into VisualApplets. Three MEDIUM RGB configurations of the Camera Link standard are supported.



**Definition of "Tap"**

The meaning of **one tap**, as used in this manual, is one decoded color component of any bit width.

Note that in the CameraLink Specification Version 2.1, **one tap** is defined as one color pixel with all color components included.

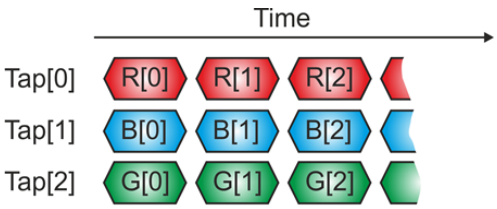
The following table shows the mapping between the VisualApplets notation and the CameraLink Specification Version 2.1:

| VA notation | CameraLink notation |
|---|---|
| Tap3x10 ..... | 10-Bit RGB Medium 1 tap |
| Tap3x12 ..... | 12-Bit RGB Medium 1 tap |
| Tap6x8 ..... | 8-Bit RGB Medium 2 taps |

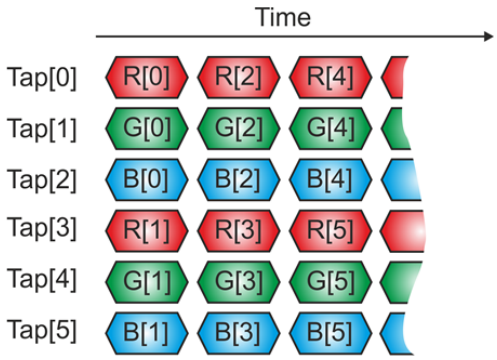
Table 29.3. Mapping VA notation and CL Specification Version 2.1

The following figures show the pixel mapping in these modes:

Tap3x10bit and Tap3x12bit:




Tap6x8bit:



The operator uses 2 ressources of type CameraPort exclusively. Thus, both ports of the frame grabber, port A (0) and port B (1), are occupied and cannot be used by other operators. The same ressource index can only be used once per applet.

| Available for Hardware Platforms |
|---|
| mE5 marathon VCLx |
| mE5 marathon VCL |
| LightBridge VCL |
| mE5 VD8-PoCL (ironman) |

### 29.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

### 29.8.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | [3,63]❶ |
| **Arithmetic** | unsigned |
| **Parallelism** | Any, see parameter description |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D,VALT_LINE1D} |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | Any |
| **Max. Img Height** | Any |

❶ The bitwidth must be a multiple of 3 due to the RGB tuple.

Parameter BitWidth can be set to a larger or smaller value than the actual bitwidth sent by the camera.

If the parameter is set to a smaller value than the actual bitwidth sent by the camera, the least significant bits (LSB) of the pixel components are cut off. Only the most signicicant bits (MSB) per pixel component are transferred into VisualApplets.

If the parameter is set to a bigger value than the actual bitwidth sent by the camera, the original pixel components are set on the MSB position of the outgoing pixel components. The LSBs are filled with zeros.

In both cases, the relative brightness stays the same, i.e., white pixels remain withe and dark pixels remain dark.

### 29.8.3. Parameters

| DvalMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |

| DvalMode |
|---|
| The parameter specifies whether DVAL is used to mask valid pixels or if it is ignored. When DVAL is ignored, the valid pixels are selected according to LVAL and FVAL statuses. |

| FvalMode | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | CameraLinkStandard |
| **Range** | {CameraLinkStandard, IgnoreFval} |

- CameraLinkStandard

  The operator expects frame valid (FVAL) signals from the camera. In this case, the operator will output frames according to the FVAL size.

- IgnoreFval

  If this setting is used, the operator will ignore any FVAL signals from the camera. All input lines will be used and merged into a sigle image with infinite height.



**Only available for Output Image Protocol VALT_IMAGE2D**

This parameter is only available if the output is set to image protocol VALT_IMAGE2D. If VALT_LINE1D is selected, the parameter is disabled and the FVAL is ignored anyway.



**VisualApplets Link Rules Violation**

When the camera operator outputs a frame of infinite height, the Max. Img Height defined in the link properties might be exceeded. This causes a violation of the VisualApplets link rules. Ensure to use valid heights. Use operator *SplitImage* to limit the image height and split the endless image into multiple chunks.

The intention of this parameter is to allow the usage of area scan an line scan cameras dynamically in the same applet.

| MediumMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Tap3x10bit |
| **Range** | {Tap3x10bit, Tap3x12bit, Tap6x8bit} |
| The parameter specifies the MEDIUM configuration mode. | |

| CameraStatus | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | LightBridge/marathon: [0;2^24-1]<br>mE5 VD8 ironman: [0;2^8-1] |

Parameter CameraStatus depends on the frame grabber hardware you are programming for. Find below the descriptions first for marathon and Lightbridge, than for mE5 VD8-PoCL (ironman).

**marathon and Lightbridge:**

bit[0]: PCLK [0] available

bit[1]: FVAL [0] current value

bit[2]: LVAL [0] current value

| CameraStatus |
|---|
| bit[3]: DVAL [0] current value<br>bit[4-11]: reserved<br>bit[12]: PCLK [1] available<br>bit[13]: FVAL [1] current value<br>bit[14]: LVAL [1] current value<br>bit[15]: DVAL [1] current value<br>bit[16-23]: reserved<br>**mE5 VD8-PoCL (ironman):**<br>bit[0]: PCLK [0] available<br>bit[1]: FVAL [0] current value<br>bit[2]: LVAL [0] current value<br>bit[3]: DVAL [0] current value<br>bit[4]: PCLK [1] available<br>bit[5]: FVAL [1] current value<br>bit[6]: LVAL [1] current value<br>bit[7]: DVAL [1] current value |

| CameraLinkCoreReset | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0; 1] |
| Reserved for support issues. | |

| PixelClockX | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel X in MHz. | |

| PixelClockY | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel Y in MHz. | |

| MinimalParallelism | |
|---|---|
| **Type** | static read parameter |
| **Default** | 2 |
| **Range** | [1; 1024] |
| Minimal parallelism for the output link O to be able to transport the maximal bandwidth of the camera without losing data. This value depends on the currently selected design clock frequency. The higher the frequency the lower the parallelism value can become. | |

# 29.9. Operator FullGrayCamera

**Operator Library: Hardware Platform**

The operator transfers image data from a Camera Link FULL GRAY configuration camera into VisualApplets. Any FULL GRAY configurations of the Camera Link standard are supported.

The operator uses 2 resources of type CameraPort exclusively. Thus, both ports of the frame grabber, port A (0) and port B (1), are occupied and cannot be used by other operators. The same resource index can only be used once per applet.

**Compatibility to VisualApplets 2.1:** In case you designed applets for mE5VD8-PoCL containing the FullGrayCamera operator with an older VisualApplets version: In earlier VisualApplets versions, operator FullGrayCamera used 3 CameraPort resources. You can load these old applets in VisualApplets 2.2. The applet gets adapted immediately, and only 2 CameraPort resources are displayed. When saving the applet in VisualApplets 2.2, the contained FullGrayCamera it is saved in its new version. After saving, the updated applet can only be loaded in VisualApplets version 2.2 or higher.

| Available for Hardware Platforms |
|---|
| mE5 VD8-PoCL (ironman) |
| mE5 marathon VCLx |
| mE5 marathon VCL |
| LightBridge VCL |

## 29.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.9.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | Any❶ |
| **Arithmetic** | unsigned |
| **Parallelism** | Any, see parameter description |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D,VALT_LINE1D} |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | Any |
| **Max. Img Height** | Any |

❶ Parameter BitWidth can be set to a larger or smaller value than the actual bitwidth sent by the camera.

If the parameter is set to a smaller value than the actual bitwidth sent by the camera, the least significant bits (LSB) of the pixel are cut off. Only the most signicicant bits (MSB) are transferred into VisualApplets.

If the parameter is set to a bigger value than the actual bitwidth sent by the camera, the original pixel is set on the MSB position. The LSBs are filled with zeros.

In both cases, the relative brightness stays the same, i.e., white pixels remain withe and dark pixels remain dark.

### 29.9.3. Parameters

| DvalMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |
| The parameter specifies whether DVAL is used to mask valid pixels or if it is ignored. When DVAL is ignored, the valid pixels are selected according to LVAL and FVAL statuses. | |

| FvalMode | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | CameraLinkStandard |
| **Range** | {CameraLinkStandard, IgnoreFval} |

- CameraLinkStandard

  The operator expects frame valid (FVAL) signals from the camera. In this case, the operator will output frames according to the FVAL size.

- IgnoreFval

  If this setting is used, the operator will ignore any FVAL signals from the camera. All input lines will be used and merged into a sigle image with infinite height.



**Only available for Output Image Protocol VALT_IMAGE2D**

This parameter is only available if the output is set to image protocol VALT_IMAGE2D. If VALT_LINE1D is selected, the parameter is disabled and the FVAL is ignored anyway.



**VisualApplets Link Rules Violation**

When the camera operator outputs a frame of infinite height, the Max. Img Height defined in the link properties might be exceeded. This causes a violation of the VisualApplets link rules. Ensure to use valid heights. Use operator *SplitImage* to limit the image height and split the endless image into multiple chunks.

The intention of this parameter is to allow the usage of area scan an line scan cameras dynamically in the same applet.

| FullMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Tap8x8bit |
| **Range** | {Tap8x8bit, Tap10x8bit, Tap8x10bit} |
| The parameter specifies the FULL configuration mode.<br><br>Tap10x8bit and Tap8x10bit are FULL/80bit configurations as specified in the CameraLink specification 2.0. | |

| CameraStatus | |
|---|---|
| **Type** | dynamic read parameter |

| CameraStatus | |
|---|---|
| **Default** | |
| **Range** | LightBridge/marathon: [0;2^36-1]<br>mE5 VD8 ironman: [0;2^12-1] |

Parameter CameraStatus depends on the frame grabber hardware you are programming for. Find below the descriptions first for marathon and Lightbridge, than for mE5 VD8-PoCL (ironman).

**marathon and Lightbridge:**

bit[0]: PCLK [0] available

bit[1]: FVAL [0] current value

bit[2]: LVAL [0] current value

bit[3]: DVAL [0] current value

bit[4-11]: reserved

bit[12]: PCLK [1] available

bit[13]: FVAL [1] current value

bit[14]: LVAL [1] current value

bit[15]: DVAL [1] current value

bit[16-23]: reserved

bit[24]: PCLK [2] available

bit[25]: FVAL [2] current value

bit[26]: LVAL [2] current value

bit[27]: DVAL [2] current value

bit[28-35]: reserved

**mE5 VD8-PoCL (ironman):**

bit[0]: PCLK [0] available

bit[1]: FVAL [0] current value

bit[2]: LVAL [0] current value

bit[3]: DVAL [0] current value

bit[4]: PCLK [1] available

bit[5]: FVAL [1] current value

bit[6]: LVAL [1] current value

bit[7]: DVAL [1] current value

bit[8]: PCLK [2] available

bit[9]: FVAL [2] current value

bit[10]: LVAL [2] current value

bit[11]: DVAL [2] current value

| CameraLinkCoreReset | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0; 1] |
| Reserved for support issues. | |

| PixelClockX | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel X in MHz. | |

| PixelClockY | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel Y in MHz. | |

| PixelClockZ | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel Z in MHz. | |

| MinimalParallelism | |
|---|---|
| **Type** | static read parameter |
| **Default** | 7 |
| **Range** | {1; 1024} |
| Minimal parallelism for the output link O to be able to transport the maximal bandwidth of the camera without losing data. This value depends on the currently selected design clock frequency. The higher the frequency the lower the parallelism value can become. | |

### 29.9.4. Examples of Use

The use of operator FullGrayCamera is shown in the following examples:

- Section 10.1.1.5, 'Grayscale Camera Link Full Area'

  Tutorial - Basic Acquisition

# 29.10. Operator FullRgbCamera

**Operator Library: Hardware Platform**

The operator transfers image data from a Camera Link FULL based RGB configuration camera into VisualApplets. Supported are three RGB modes. One mode, Tap8x8bit, is not standard conform.



**Definition of "Tap"**

The meaning of **one tap**, as used in this document, is one decoded color component of any bit width.

Note that in the CameraLink Specification Version 2.1, **one tap** is defined as one color pixel with all color components included.

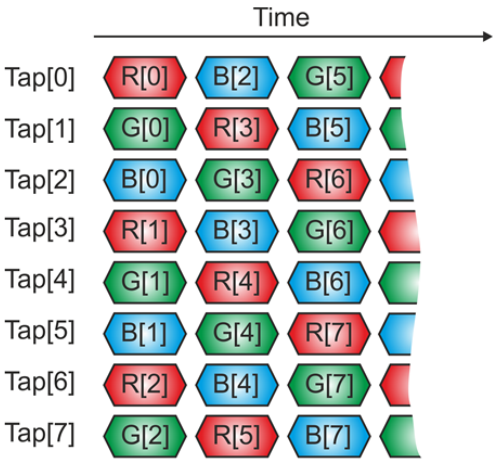
The following table shows the mapping between the VA notation and the CameraLink Specification Version 2.1

| VA notation | CameraLink notation |
|---|---|
| Tap8x8bit ..... | not covered by the CL Specification, used by selected cameras |
| Tap10x8bit ..... | 8-Bit, 80-bit, RGB 10-tap |
| Tap8x10bit ..... | 10-Bit, 80-bit RGB 8-tap |

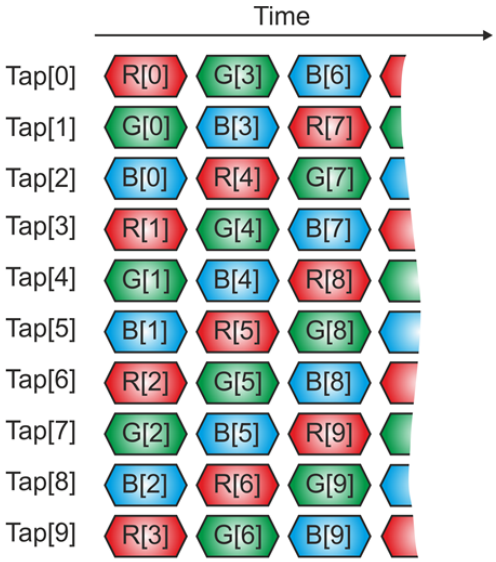
Table 29.4. Mapping VA notation and CL Specification Version 2.1

The following figures show the pixel mapping in these modes:

Tap8x8bit and Tap8x10bit:




Tap10x8bit:




The operator uses 2 ressources of type CameraPort exclusively. Thus, both ports of the frame grabber, port A (0) and port B (1), are occupied and cannot be used by other operators. The same ressource index can only be used once per applet.

| Available for Hardware Platforms |
|---|
| mE5 marathon VCLx |
| mE5 marathon VCL |
| LightBridge VCL |
| mE5 VD8-PoCL (ironman) |

### 29.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

### 29.10.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | [3,63]❶ |
| **Arithmetic** | unsigned |
| **Parallelism** | Any, see parameter description |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D,VALT_LINE1D} |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | Any |
| **Max. Img Height** | Any |

❶ The bitwidth must be a multiple of 3 due to the RGB tuple.

Parameter BitWidth can be set to a larger or smaller value than the actual bitwidth sent by the camera.

If the parameter is set to a smaller value than the actual bitwidth sent by the camera, the least significant bits (LSB) of the pixel components are cut off. Only the most signicicant bits (MSB) per pixel component are transferred into VisualApplets.

If the parameter is set to a bigger value than the actual bitwidth sent by the camera, the original pixel components are set on the MSB position of the outgoing pixel components. The LSBs are filled with zeros.

In both cases, the relative brightness stays the same, i.e., white pixels remain withe and dark pixels remain dark.

### 29.10.3. Parameters

| **DvalMode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |
| The parameter specifies whether DVAL is used to mask valid pixels or if it is ignored. When DVAL is ignored, the valid pixels are selected according to LVAL and FVAL statuses. | |

| **FvalMode** | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | CameraLinkStandard |
| **Range** | {CameraLinkStandard, IgnoreFval} |

- CameraLinkStandard

  The operator expects frame valid (FVAL) signals from the camera. In this case, the operator will output frames according to the FVAL size.

- IgnoreFval

  If this setting is used, the operator will ignore any FVAL signals from the camera. All input lines will be used and merged into a sigle image with infinite height.



**Only available for Output Image Protocol VALT_IMAGE2D**

This parameter is only available if the output is set to image protocol VALT_IMAGE2D. If VALT_LINE1D is selected, the parameter is disabled and the FVAL is ignored anyway.



**VisualApplets Link Rules Violation**

When the camera operator outputs a frame of infinite height, the Max. Img Height defined in the link properties might be exceeded. This causes a violation of the VisualApplets link rules. Ensure to use valid heights. Use operator *SplitImage* to limit the image height and split the endless image into multiple chunks.

The intention of this parameter is to allow the usage of area scan an line scan cameras dynamically in the same applet.

| **FullMode** | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Tap8x8bit |
| **Range** | {Tap8x8bit, Tap8x10bit, Tap10x8bit} |

| **FullMode** |
| --- |
| The parameter specifies the FULL configuration mode. |

| **CameraStatus** | |
| --- | --- |
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | LightBridge/marathon: [0; 2^24-1]<br>mE5 VD8 ironman: [0;2^12-1] |

Parameter CameraStatus depends on the frame grabber hardware you are programming for. Find below the descriptions first for marathon and Lightbridge, than for mE5 VD8-PoCL (ironman).

**marathon and Lightbridge:**

bit[0]: PCLK [0] available

bit[1]: FVAL [0] current value

bit[2]: LVAL [0] current value

bit[3]: DVAL [0] current value

bit[4..7]: reserved

bit[8]: PCLK [1] available

bit[9]: FVAL [1] current value

bit[10]: LVAL [1] current value

bit[11]: DVAL [1] current value

bit[12..15]: reserved

bit[16]: PCLK [2] available

bit[17]: FVAL [2] current value

bit[18]: LVAL [2] current value

bit[19]: DVAL [2] current value

bit[20..23]: reserved

**mE5 VD8-PoCL (ironman):**

bit[0]: PCLK [0] available

bit[1]: FVAL [0] current value

bit[2]: LVAL [0] current value

bit[3]: DVAL [0] current value

bit[4]: PCLK [1] available

bit[5]: FVAL [1] current value

bit[6]: LVAL [1] current value

bit[7]: DVAL [1] current value

bit[8]: PCLK [2] available

bit[9]: FVAL [2] current value

| CameraStatus |
|---|
| bit[10]: LVAL [2] current value<br><br>bit[11]: DVAL [2] current value |

| CameraLinkCoreReset | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0; 1] |
| Reserved for support issues. | |

| PixelClockX | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel X in MHz. | |

| PixelClockY | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel Y in MHz. | |

| PixelClockZ | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0; 85] MHz |
| Pixel clock rate of camera channel Z in MHz. | |

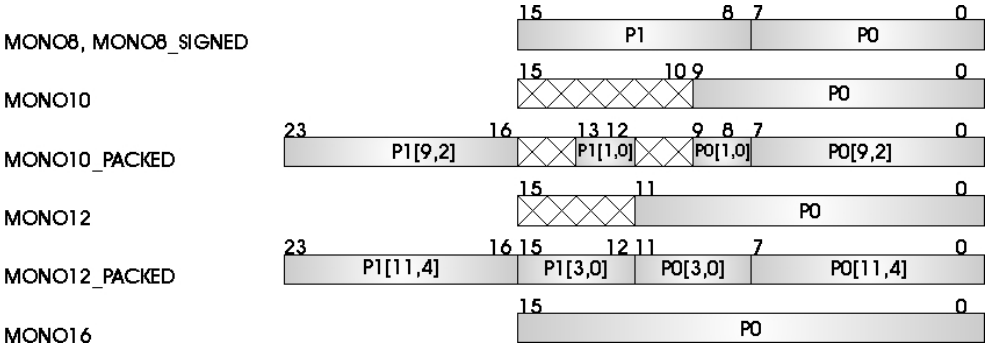
| MinimalParallelism | |
|---|---|
| **Type** | static read parameter |
| **Default** | 3 |
| **Range** | [1; 1024] |
| Minimal parallelism for the output link O to be able to transport the maximal bandwidth of the camera without losing data. This value depends on the currently selected design clock frequency. The higher the frequency the lower the parallelism value can become. | |

# 29.11. Operator CameraGrayArea

**Operator Library: Hardware Platform**

This operator represents the image data interface between a grayscale area scan GigE Vision camera and VisualApplets. This operator requires one VisualApplets resource of type **CAM**. Set the resource index for the camera in the resource dialog. The resource index 0 maps the camera to port 0, index 1 to port 1, etc. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support the different data formats specified by the GigE Vision standard. The following figure illustrates the data formats.




The packed formats require 3 bytes for transmitting 2 pixels. MONO8 and MONO8_SIGNED require only 1 byte per pixel. MONO10, MONO12 and MONO16 need 2 bytes per pixel.

**Note**

- MONO8_SIGNED is managed by the operator as MONO8 and the output link format will be set to UNSIGNED type. If a SIGNED link is required use CastType operator and change the link type to SIGNED.
- Bayer formats can be mapped to these Gray formats because the camera operator does not interpret the pixel content.

The parameterized format is converted to match with the parameterized output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum.

| Available for Hardware Platforms |
|---|
| microEnable IV VQ4-GE/-GPoE |

## 29.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.11.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {8, 10, 12, 14, 16} |

| Link Parameter | Output Link O |
|---|---|
| **Arithmetic** | unsigned |
| **Parallelism** | 4 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE2D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.11.3. Parameters

| PixelFormat | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | MONO8 |
| **Range** | {MONO8, MONO8_SIGNED, MONO10, MONO10_PACKED, MONO12, MONO12_PACKED, MONO16} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

| Overflow | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0, 1} |
| This parameter indicates if an overflow at the camera operator has occurred i.e. image data has been lost. Each time an overflow occurs, the parameter is set to one. This value is kept until the value is read. Thus, reading the parameter will reset it to value 0. In normal operations this should never happen. The parameter is used for maintenance. | |

### 29.11.4. Examples of Use

The use of operator CameraGrayArea is shown in the following examples:

- Section 10.2.1.1, ' GigE Vision Grayscale Area Scan Cameras '

  Tutorial - Basic Acquisition

# 29.12. Operator CameraGrayAreaBase

**Operator Library: Hardware Platform**

This operator represents the image data interface between a grayscale area scan camera in Camera Link base configuration mode and VisualApplets. This operator requires one VisualApplets resource of type **CAM**. Set the resource index for the camera in the resource dialog. The resource index 0 maps the camera to port A and the resource index 1 maps to the camera to port B. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support different data formats specified by the CamerLink standard. Use parameter *Format* to set the camera operator to the same mode as used by the camera.

The parameterized format is converted to match with the output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.12.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {8, 10, 12, 14, 16} |
| **Arithmetic** | unsigned |
| **Parallelism** | 4 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE2D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.12.3. Parameters

| UseDval | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |

| UseDval |
|---|
| By setting this parameter to DVAL_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional DVAL signal is used. In general users should not modify this parameter! |

| Format | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | SingleTap8bit |
| **Range** | {SingleTap8bit, SingleTap10bit, SingleTap12bit, SingleTap14bit, SingleTap16bit} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

## 29.12.4. Examples of Use

The use of operator CameraGrayAreaBase is shown in the following examples:

- 2. *Getting Started*

  Getting Started

- Figure 3.1, 'Simple VisualApplets Design'

  Basic Principles - Learn the Idea of VisualApplets

- Section 3.5, 'Data Flow '

  Data Flow - Learn about the Pipeline Structure used in VisualApplets

- Section 3.6.9, 'Infinite Sources / Connecting Cameras'

  Infinite Sources - Connecting operators to cameras. (DRC2 Latency Error)

- Section 3.8, 'Allocation of Device Resources'

  Learn the allocation of the device resources of the operator.

- Section 9.1, 'Applet Parameterization'

  Tutorial Basic Acquisition - Camera Operator Parameterization

- Section 10.1.1.1, 'Grayscale Camera Link Base Area'

  Tutorial - Basic Acquisition

# 29.13. Operator CameraGrayAreaFull

**Operator Library: Hardware Platform**

This operator represents the image data interface between a grayscale area scan camera in Camera Link full configuration mode and VisualApplets. This operator requires two VisualApplets resource of type **CAM**. The resources are automatically occupied by the operator. Both camera ports (port A and port B) are used by this operator. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support different data formats specified by the CamerLink standard. Use parameter *FullMode* to set the camera operator to the same mode as used by the camera. Moreover, the link can be set to different bit widths. The following table shows allowed combinations of Parameter FullMode, its access type, the bit width and parallelism.

| FullMode | Access to FullMode | Bit Width | Parallelism |
|---|---|---|---|
| Tap8x8bit | Dynamic/Static | 8 | 8 |
| Tap8x8bit | Dynamic/Static | 8 | 16 |
| Tap10x8bit | Dynamic/Static | 8 | 8 |
| Tap10x8bit | Dynamic/Static | 8 | 16 |
| Tap8x10bit | Static | 10 | 12 |

Note that when using a parallelism of 8, it can be possible that the datarate of the camera is higher than the available bandwidth of the operator's output link. On the microEnable IV frame grabbers, the theoretical bandwidth of the link at parallelism eight is 500MPixel/s. In CameraLink full configuration mode, up to 850MPixel/s can be transferred.

Besides the bit width and the parallelism, the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.13.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.13.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {8, 10, 12, 14, 16}❶ |
| **Arithmetic** | unsigned |
| **Parallelism** | {8, 12, 16}}❷ |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE2D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |

| Link Parameter | Output Link O |
|---|---|
| **Max. Img Height** | any |

⓬ Not all combinations between the bit width, the parallelism and parameter settings are allowed. See table above.

## 29.13.3. Parameters

| DvalMode | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |
| By setting this parameter to DVAL_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional DVAL signal is used. In general users should not modify this parameter! | |

| FullMode | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Tap8x8bit |
| **Range** | {Tap8x8bit, Tap10x8bit, Tap8x10bit} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration.<br><br>Not all combinations between the bit width, the parallelism and parameter settings are allowed. See table above. | |

## 29.13.4. Examples of Use

The use of operator CameraGrayAreaFull is shown in the following examples:

- Section 10.1.1.5, 'Grayscale Camera Link Full Area'

  Tutorial - Basic Acquisition

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

# 29.14. Operator CameraGrayAreaMedium

**Operator Library: Hardware Platform**

This operator represents the image data interface between a grayscale area scan camera in Camera Link madium configuration mode and VisualApplets. This operator requires two VisualApplets resource of type **CAM**. The resources are automatically occupied by the operator. Both camera ports (port A and port B) are used by this operator. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support different data formats specified by the CamerLink standard. Use parameter *Format* to set the camera operator to the same mode as used by the camera.

The parameterized format is converted to match with the output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.14.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.14.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {8, 10, 12} |
| **Arithmetic** | unsigned |
| **Parallelism** | {4, 8}❶ |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE2D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

❶ Note that when using a parallelism of 4, it can be possible that the datarate of the camera is higher than the available bandwidth of the operator's output link. On the microEnable IV frame grabbers, the theoretical bandwidth of the link at parallelism four is 250MPixel/s. In CameraLink medium configuration mode, up to 340MPixel/s can be transferred.

### 29.14.3. Parameters

| UseDval | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | DVAL_Enabled |
| **Range** | {DVAL_Enabled, DVAL_Disabled} |
| By setting this parameter to DVAL_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional DVAL signal is used. In general users should not modify this parameter! | |

| Format | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | QuadTap8Bit |
| **Range** | {QuadTap8Bit, QuadTap10Bit, QuadTap12Bit} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

### 29.14.4. Examples of Use

The use of operator CameraGrayAreaMedium is shown in the following examples:

- Section 10.1.1.3, 'Grayscale Camera Link Medium Area'

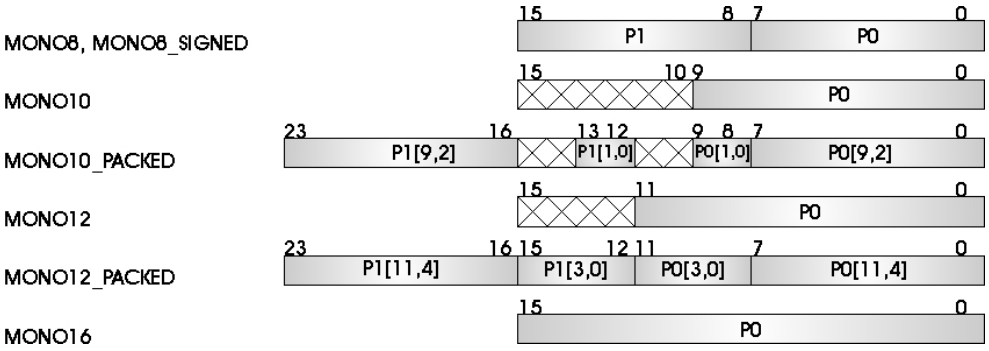
  Tutorial - Basic Acquisition

# 29.15. Operator CameraGrayLine

**Operator Library: Hardware Platform**

This operator represents the image data interface between a grayscale line scan GigE Vision camera and VisualApplets. This operator requires one VisualApplets resource of type **CAM**. Set the resource index for the camera in the resource dialog. The resource index 0 maps the camera to port 0, index 1 to port 1, etc. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support the different data formats specified by the GigE Vision standard. The following figure illustrates the data formats.




The packed formats require 3 bytes for transmitting 2 pixels. MONO8 and MONO8_SIGNED require only 1 byte per pixel. MONO10, MONO12 and MONO16 need 2 bytes per pixel.

**Note**

- MONO8_SIGNED is managed by the operator as MONO8 and the output link format will be set to UNSIGNED type. If a SIGNED link is required use CastType operator and change the link type to SIGNED.
- Bayer formats can be mapped to these Gray formats because the camera operator does not interpret the pixel content.

The parameterized format is converted to match with the parameterized output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum. For the 1D image protocol, the height is ignored.

| Available for Hardware Platforms |
|---|
| microEnable IV VQ4-GE/-GPoE |

## 29.15.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.15.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {8, 10, 12, 14, 16} |

| Link Parameter | Output Link O |
|---|---|
| **Arithmetic** | unsigned |
| **Parallelism** | 4 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.15.3. Parameters

| **PixelFormat** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | MONO8 |
| **Range** | {MONO8, MONO8_SIGNED, MONO10, MONO10_PACKED, MONO12, MONO12_PACKED, MONO16} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

### 29.15.4. Examples of Use

The use of operator CameraGrayLine is shown in the following examples:

- Section 10.2.2.1, 'GigE Vision Grayscale Line Scan Cameras'

  Tutorial - Basic Acquisition

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

# 29.16. Operator CameraGrayLineBase

**Operator Library: Hardware Platform**

This operator represents the image data interface between a grayscale line scan camera in Camera Link base configuration mode and VisualApplets. This operator requires one VisualApplets resource of type **CAM**. Set the resource index for the camera in the resource dialog. The resource index 0 maps the camera to port A and the resource index 1 maps to the camera to port B. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support different data formats specified by the CamerLink standard. Use parameter *Format* to set the camera operator to the same mode as used by the camera.

The parameterized format is converted to match with the output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum. For the 1D image protocol, the height is ignored.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.16.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.16.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {8, 10, 12, 14, 16} |
| **Arithmetic** | unsigned |
| **Parallelism** | 4 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.16.3. Parameters

| UseDval | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Dval_Enabled |

| UseDval | |
|---|---|
| **Range** | {Dval_Enabled, Dval_Disabled} |
| By setting this parameter to Dval_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional Dval signal is used. In general users should not modify this parameter! | |

| Format | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | SingleTap8bit |
| **Range** | {SingleTap8bit, SingleTap10bit, SingleTap12bit, SingleTap14bit, SingleTap16bit} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

### 29.16.4. Examples of Use

The use of operator CameraGrayLineBase is shown in the following examples:

- Section 3.8, 'Allocation of Device Resources'

  Learn the allocation of the device resources of the operator.

- Section 10.1.2.1, 'Grayscale Camera Link Base Line'

  Tutorial - Basic Acquisition

- Section 10.3.2.1, 'Grayscale Camera Link Base Line Scan Cameras '

  Tutorial - Basic Acquisition

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 29.17. Operator CameraGrayLineFull

**Operator Library: Hardware Platform**

This operator represents the image data interface between a grayscale line scan camera in Camera Link full configuration mode and VisualApplets. This operator requires two VisualApplets resource of type **CAM**. The resources are automatically occupied by the operator. Both camera ports (port A and port B) are used by this operator. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support different data formats specified by the CamerLink standard. Use parameter *FullMode* to set the camera operator to the same mode as used by the camera. Moreover, the link can be set to different bit widths. The following table shows allowed combinations of Parameter FullMode, its access type, the bit width and parallelism.

| FullMode | Access to FullMode | Bit Width | Parallelism |
|---|---|---|---|
| Tap8x8bit | Dynamic/Static | 8 | 8 |
| Tap8x8bit | Dynamic/Static | 8 | 16 |
| Tap10x8bit | Dynamic/Static | 8 | 8 |
| Tap10x8bit | Dynamic/Static | 8 | 16 |
| Tap8x10bit | Static | 10 | 12 |

Note that when using a parallelism of 8, it can be possible that the datarate of the camera is higher than the available bandwidth of the operator's output link. On the microEnable IV frame grabbers, the theoretical bandwidth of the link at parallelism eight is 500MPixel/s. In CameraLink full configuration mode, up to 850MPixel/s can be transferred.

Besides the bit width and the parallelism, the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum. For the 1D image protocol, the height is ignored.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.17.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.17.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {8, 10, 12, 14, 16}❶ |
| **Arithmetic** | unsigned |
| **Parallelism** | {8, 12, 16}}❷ |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |

| Link Parameter | Output Link O |
|---|---|
| **Max. Img Height** | any |

⓬ Not all combinations between the bit width, the parallelism and parameter settings are allowed. See table above.

## 29.17.3. Parameters

| DvalMode | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Dval_Enabled |
| **Range** | {Dval_Enabled, Dval_Disabled} |
| By setting this parameter to Dval_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional Dval signal is used. In general users should not modify this parameter! | |

| FullMode | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Tap8x8bit |
| **Range** | {Tap8x8bit, Tap10x8bit, Tap8x10bit} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration.<br><br>Not all combinations between the bit width, the parallelism and parameter settings are allowed. See table above. | |

## 29.17.4. Examples of Use

The use of operator CameraGrayLineFull is shown in the following examples:

- Section 10.1.2.4, 'Grayscale Camera Link Full Line'

  Tutorial - Basic Acquisition

- Section 10.3.2.5, 'Grayscale Camera Link Full Line Scan Cameras '

  Tutorial - Basic Acquisition

# 29.18. Operator CameraGrayLineMedium

**Operator Library: Hardware Platform**

This operator represents the image data interface between a grayscale line scan camera in Camera Link madium configuration mode and VisualApplets. This operator requires two VisualApplets resource of type **CAM**. The resources are automatically occupied by the operator. Both camera ports (port A and port B) are used by this operator. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support different data formats specified by the CamerLink standard. Use parameter *Format* to set the camera operator to the same mode as used by the camera.

The parameterized format is converted to match with the output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum. For the 1D image protocol, the height is ignored.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.18.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.18.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {8, 10, 12} |
| **Arithmetic** | unsigned |
| **Parallelism** | {4, 8}❶ |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE1D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

❶ Note that when using a parallelism of 4, it can be possible that the datarate of the camera is higher than the available bandwidth of the operator's output link. On the microEnable IV frame grabbers, the theoretical bandwidth of the link at parallelism four is 250MPixel/s. In CameraLink medium configuration mode, up to 340MPixel/s can be transferred.

### 29.18.3. Parameters

| UseDval | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Dval_Enabled |
| **Range** | {Dval_Enabled, Dval_Disabled} |
| By setting this parameter to Dval_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional Dval signal is used. In general users should not modify this parameter! | |

| Format | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | QuadTap8Bit |
| **Range** | {QuadTap8Bit, QuadTap10Bit, QuadTap12Bit} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

### 29.18.4. Examples of Use

The use of operator CameraGrayLineMedium is shown in the following examples:

- Section 10.1.2.3, 'Grayscale Camera Link Medium Line'

  Tutorial - Basic Acquisition

- Section 10.3.2, 'Camera Link Line Scan Cameras '

  Tutorial - Basic Acquisition

- Section 10.3.2.3, 'Grayscale Camera Link Medium Line Scan Cameras '

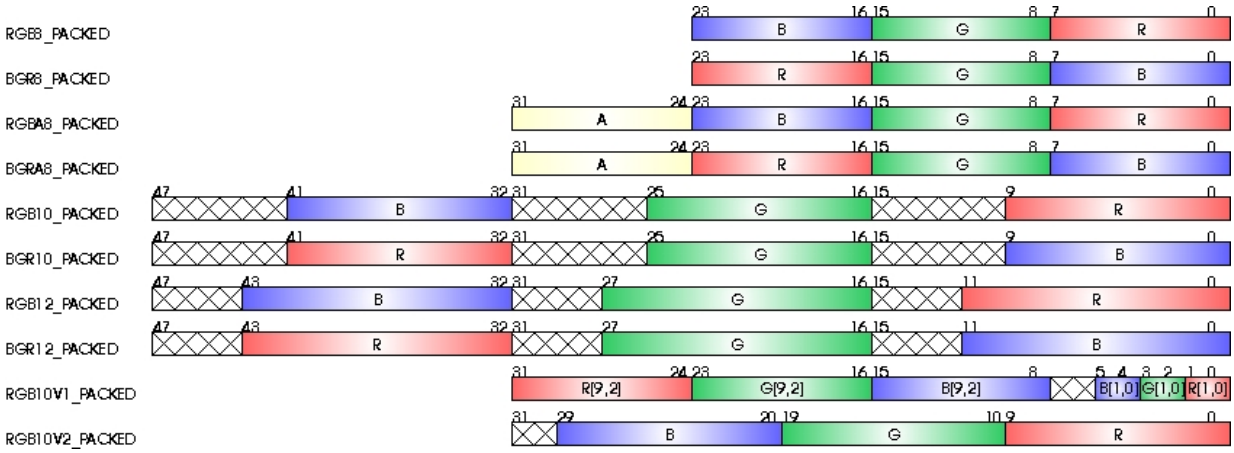
  Tutorial - Basic Acquisition

# 29.19. Operator CameraRgbArea

**Operator Library: Hardware Platform**

This operator represents the image data interface between a RGB area scan GigE Vision camera and VisualApplets. This operator requires one VisualApplets resource of type **CAM**. Set the resource index for the camera in the resource dialog. The resource index 0 maps the camera to port 0, index 1 to port 1, etc. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support the different data formats specified by the GigE Vision standard. The following figure illustrates the data formats.




**Note**

- RGBA8_PACKED and BGRA8_PACKED transmit an alpha component. However the camera port link is specified as RGB. Therefore the alpha component (the 4th component) is cut off inside the camera port and only the other three R, G and B components are provided on the output link. If the 4th component is required, use CameraGrayArea and set the format to MONO8. The 4 provided pixels will build a single RGBA pixel. Now reinterpret the gray pixels as required.

- The camera operator extracts and aligns pixels as needed. For example setting the format to BGR8_PACKED will force the camera operator to swap B and R components internally. At the output link a perfect RGB pixel is provided. The same applies for RGB10V1 and RGB10V2 formats. The camera operator will output 3 concatenated components.

The parameterized format is converted to match with the parameterized output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum.

| Available for Hardware Platforms |
|---|
| microEnable IV VQ4-GE/-GPoE |

## 29.19.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

### 29.19.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {24, 30, 36} |
| **Arithmetic** | unsigned |
| **Parallelism** | {2, 4} |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE2D |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.19.3. Parameters

| PixelFormat | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | RGB8_PACKED |
| **Range** | {RGB8_PACKED, BGR8_PACKED, RGBA8_PACKED, BGRA8_PACKED, RGB10_PACKED, BGR10_PACKED, RGB12_PACKED, BGR12_PACKED, RGB10V1_PACKED, RGB10V2_PACKED} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

### 29.19.4. Examples of Use

The use of operator CameraRgbArea is shown in the following examples:

- Section 10.2.1.2, ' GigE Vision RGB Area Scan Cameras '

  Tutorial - Basic Acquisition

# 29.20. Operator CameraRgbAreaBase

**Operator Library: Hardware Platform**

This operator represents the image data interface between a RGB area scan camera in RGB Camera Link base configuration mode and VisualApplets. This operator requires one VisualApplets resource of type **CAM**. Set the resource index for the camera in the resource dialog. The resource index 0 maps the camera to port A and the resource index 1 maps to the camera to port B. Check Section 3.8, 'Allocation of Device Resources' for more information.

In the output link, the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.20.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.20.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 24 |
| **Arithmetic** | unsigned |
| **Parallelism** | 2 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE2D |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.20.3. Parameters

| UseDval | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Dval_Enabled |
| **Range** | {Dval_Enabled, Dval_Disabled} |
| By setting this parameter to Dval_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional Dval signal is used. In general users should not modify this parameter! | |

## 29.20.4. Examples of Use

The use of operator CameraRgbAreaBase is shown in the following examples:

- Section 12.4.4, 'RGB White Balancing'

  Examples - The applet shows an example for white balancing on RGB images.

# 29.21. Operator CameraRgbAreaMedium

**Operator Library: Hardware Platform**

This operator represents the image data interface between a RGB area scan camera in Camera Link RGB medium configuration mode and VisualApplets. This operator requires two VisualApplets resource of type **CAM**. The resources are automatically occupied by the operator. Both camera ports (port A and port B) are used by this operator. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support different data formats specified by the CamerLink standard. Use parameter *Format* to set the camera operator to the same mode as used by the camera.

The parameterized format is converted to match with the output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.21.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.21.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {30, 36} |
| **Arithmetic** | unsigned |
| **Parallelism** | {2, 3, 4} |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE2D |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.21.3. Parameters

| UseDval | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Dval_Enabled |
| **Range** | {Dval_Enabled, Dval_Disabled} |

| UseDval |
| --- |
| By setting this parameter to Dval_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional Dval signal is used. In general users should not modify this parameter! |

| Format | |
| --- | --- |
| **Type** | dynamic read/write parameter |
| **Default** | RGB30 |
| **Range** | {RGB30, RGB36} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

## 29.21.4. Examples of Use

The use of operator CameraRgbAreaMedium is shown in the following examples:

- Section 10.1.1.4, 'RGB Camera Link Medium Area'

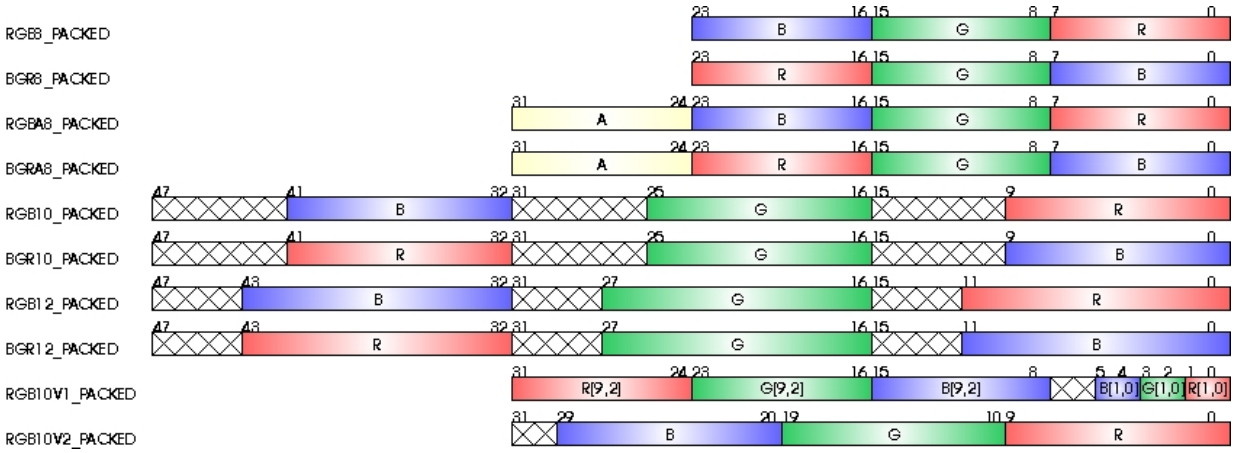
  Tutorial - Basic Acquisition

# 29.22. Operator CameraRgbLine

**Operator Library: Hardware Platform**

This operator represents the image data interface between a RGB line scan GigE Vision camera and VisualApplets. This operator requires one VisualApplets resource of type **CAM**. Set the resource index for the camera in the resource dialog. The resource index 0 maps the camera to port 0, index 1 to port 1, etc. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support the different data formats specified by the GigE Vision standard. The following figure illustrates the data formats.




**Note**

- RGBA8_PACKED and BGRA8_PACKED transmit an alpha component. However the camera port link is specified as RGB. Therefore the alpha component (the 4th component) is cut off inside the camera port and only the other three R, G and B components are provided on the output link. If the 4th component is required, use CameraGrayArea and set the format to MONO8. The 4 provided pixels will build a single RGBA pixel. Now reinterpret the gray pixels as required.

- The camera operator extracts and aligns pixels as needed. For example setting the format to BGR8_PACKED will force the camera operator to swap B and R components internally. At the output link a perfect RGB pixel is provided. The same applies for RGB10V1 and RGB10V2 formats. The camera operator will output 3 concatenated components.

The parameterized format is converted to match with the parameterized output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum.

| Available for Hardware Platforms |
|---|
| microEnable IV VQ4-GE/-GPoE |

## 29.22.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

### 29.22.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {24, 30, 36} |
| **Arithmetic** | unsigned |
| **Parallelism** | {2, 4} |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE1D |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.22.3. Parameters

| **PixelFormat** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | RGB8_PACKED |
| **Range** | {RGB8_PACKED, BGR8_PACKED, RGBA8_PACKED, BGRA8_PACKED, RGB10_PACKED, BGR10_PACKED, RGB12_PACKED, BGR12_PACKED, RGB10V1_PACKED, RGB10V2_PACKED} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

# 29.23. Operator CameraRgbLineBase

**Operator Library: Hardware Platform**

This operator represents the image data interface between a RGB line scan camera in RGB Camera Link base configuration mode and VisualApplets. This operator requires one VisualApplets resource of type **CAM**. Set the resource index for the camera in the resource dialog. The resource index 0 maps the camera to port A and the resource index 1 maps to the camera to port B. Check Section 3.8, 'Allocation of Device Resources' for more information.

In the output link, the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum. For the 1D image protocol, the height is ignored.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.23.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.23.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 24 |
| **Arithmetic** | unsigned |
| **Parallelism** | 2 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE1D |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.23.3. Parameters

| UseDval | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Dval_Enabled |
| **Range** | {Dval_Enabled, Dval_Disabled} |
| By setting this parameter to Dval_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional Dval signal is used. In general users should not modify this parameter! | |

### 29.23.4. Examples of Use

The use of operator CameraRgbLineBase is shown in the following examples:

- Section 10.1.2.2, 'RGB Camera Link Base Line'

  Tutorial - Basic Acquisition
- Section 10.3.2.2, 'RGB Camera Link Base Line Scan Cameras '

  Tutorial - Basic Acquisition

# 29.24. Operator CameraRgbLineMedium

**Operator Library: Hardware Platform**

This operator represents the image data interface between a RGB line scan camera in Camera Link RGB medium configuration mode and VisualApplets. This operator requires two VisualApplets resource of type **CAM**. The resources are automatically occupied by the operator. Both camera ports (port A and port B) are used by this operator. Check Section 3.8, 'Allocation of Device Resources' for more information.

This operator can be configured to support different data formats specified by the CamerLink standard. Use parameter *Format* to set the camera operator to the same mode as used by the camera.

The parameterized format is converted to match with the output link format of the camera operator. The output link can be configured to several bit widths. Conversion is performed by adding bits to the lower bit positions if the camera format bit width is higher than the parameterized bit width in the output link. However, if the camera format bit width is less than the available bits at the output link, only the most significant bits are used.

Moreover, in the output link the maximum possible image dimensions have to be specified. The size of the actual transfered images however may be less but must not exceed this parameterized maximum. For the 1D image protocol, the height is ignored.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.24.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, image data output |

## 29.24.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | {30, 36} |
| **Arithmetic** | unsigned |
| **Parallelism** | {2, 3, 4} |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE1D |
| **Color Format** | VAF_COLOR |
| **Color Flavor** | FL_RGB |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.24.3. Parameters

| UseDval | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | Dval_Enabled |
| **Range** | {Dval_Enabled, Dval_Disabled} |

| UseDval |
| --- |
| By setting this parameter to Dval_Disabled it is possible to support cameras which do not fully comply with the CameraLink specification. In disabled mode, control signals Lval and Fval are used to decode pixels. In enabled mode, the additional Dval signal is used. In general users should not modify this parameter! |

| Format | |
| --- | --- |
| **Type** | dynamic read/write parameter |
| **Default** | RGB30 |
| **Range** | {RGB30, RGB36} |
| This parameter specifies the data format of the connected camera. The format has to match with the camera configuration. | |

## 29.24.4. Examples of Use

The use of operator CameraRgbLineMedium is shown in the following examples:

- Section 10.3.2, 'Camera Link Line Scan Cameras '

  Tutorial - Basic Acquisition

- Section 10.3.2.4, 'RGB Camera Link Medium Line Scan Cameras '

  Tutorial - Basic Acquisition

# 29.25. Operator CLHSDualCamera

**Operator Library: Hardware Platform**

This operator represents the image data interface between a CLHS Dual Link configuration camera and VisualApplets. This operator receives data from the CLHS Dual Link configuration camera and feeds it into the image processing application.

| Available for Hardware Platform |
| --- |
| marathon VF2 |

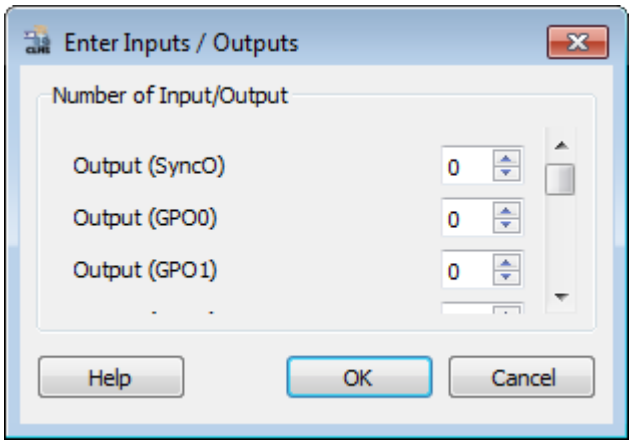
The operator uses both resources of type camera exclusively. More camera resources cannot be used in the applet.

The operator provides image data on its output port O. This output port is always present.

In addition to this standard output port, you can specify various optional ports if required by your design.

The CLHS protocoll supports pulses and GPIO messages.

During instantiation of the operator, a pop-up dialog appears:




Here, you can define the availability of up to 16 optional GPIs (General Purpose Inputs) and up to 16 optional GPOs (General Purpose Outputs).

In addition, you can activate one output port **Output (SyncO)** (MultiFGSync bit of video message).

If you set the port availability to "0" (default), the port will not be present in the operator. If you set the port availability to 1, the particular port is available at the operator interface.

The GPIO state is transferred when the state of a GPI is changed by a connected operator. When the operator receives a GPIO message from the camera, all GPOs are updated.

GPIO messages are only exchanged on channel 0. On channel 1 there is no communication of this kind.

Parameter Bit Width can be set to a value higher or lower than the bit width the camera is actually sending.

If you select a lower value, the LSB bits of each pixel are cut off, so that only the MSB bits are transferred into the application.

If you select a higher value, the original pixel is set to the MSB position of the outgoing pixel. The LSB bits are filled with zeros.

In both cases, the relative brightness remains the same, i.e., white pixels remain white and dark pixels remain dark.

### 29.25.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | GPI0...GPI15 (optional), status of the frame grabber GPOs (exchange via CLHS) |
| **Output Links** | O, acquisition image data to be used inside VisualApplets<br>GPO0...GPO15 (optional), status of the camera GPOs (exchange via CLHS)<br>SyncO, MultiFGSync bit is set in video package (valid for 1 clock cycle) |

### 29.25.2. Supported Link Format

| Link Parameter | Output Link O | Output Link GPO0...GPO15 (optional) |
|---|---|---|
| **Bit Width** | any | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | any, see parameter description | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | 1 |
| **Max. Img Height** | any | 1 |

| Link Parameter | Input Link GPI0...GPI15 (optional) | Output Link SyncO |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | 1 | 1 |
| **Max. Img Height** | 1 | 1 |

### 29.25.3. Parameters

| CameraID | |
|---|---|
| **Type** | static write parameter |

| CameraID | |
|---|---|
| **Default** | 0 |
| **Range** | 0 |
| The parameter specifies which camera resource is used. Furthermore, the ID will be used to map camera channels to applet ports. | |

| PixelFormat | |
|---|---|
| **Type** | static write parameter |
| **Default** | Mono10p |
| **Range** | {Mono8,Mono10p,Mono12p,Mono14p,Mono16,Raw8,Raw10p,Raw12p,Raw14p,Raw16, Raw64, B8, B10p, B12p, B14p, B16, G8, G10p, G12p, G14p, G16, R8, R10p, R12p, R14p, R16, BGR8, BGR10p, BGR12p, BGR14p, BGR16, BGRa8, BGRa10p, BGRa12p, BGRa14p, BGRa16, BayerGR8, BayerGR10p, BayerGR12p,BayerGR14p, BayerGR16, BayerRG8, BayerRG10p, BayerRG12p, BayerRG14p, BayerRG16, BayerGB8, BayerGB10p, BayerGB12p, BayerGB14p, BayerGB16, BayerBG8, BayerBG10p, BayerBG12p, BayerBG14p, BayerBG16} |
| The parameter specifies the output pixel format. | |

| AquisitionFormat | |
|---|---|
| **Type** | static write parameter |
| **Default** | Area |
| **Range** | {Area; Linescan} |
| When this parameter is set to "Area", the first data available is a complete frame. Frames started in the middle will be ignored. Therefor a "Start Of Frame" in the first video message is required. In "linescan" mode, Start/End of Frame will be ignored, since they are set to 0 for linescan cameras. | |

| MinimalParallelism | |
|---|---|
| **Type** | static read parameter |
| **Default** | 16 |
| **Range** | [1; 1024] |
| Minimal parallelism for the output link O to be able to transport the maximal bandwidth of the camera without losing data. This value depends on the currently selected design clock frequency. The higher the frequency the lower the parallelism value can become. | |

| LineSupervision | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | off |
| **Range** | {on, off} |
| Consistent check for lines. Checks if RowIDs following by RowStep steps. Furthermore check for integrity: Start of line (SoL) and End of Line (EoL) must be in the correct order. Missing EoL or SoL will be detected, an error flag is generated and an End of Line is inserted. Missing Lines will be inserted as smaller lines (length depends on parallelism and bitwidth) until 2 following RowIDs are detected and line consistency is guaranteed again. | |

| RowStep | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | {1, 65535} |
| Step between 2 RowIDs. Should be 1 unless a multi-channel camera is used and lines are split amongst links. | |

| OverflowOccurred | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0;1] |
| This parameter signalizes that the internal FIFO has an overflow. Overflow indicates loss of data and need to be avoided. Possible reasons might be: One link is not sending at all, or the data is received faster than it can be processed by the operator. | |

# 29.26. Operator CLHSPulseIn

**Operator Library: Hardware Platform**

This operator manages the sending of pulse messages.

| Available for Hardware Platforms |
| --- |
| microEnable 5 marathon VF2 |

The operator uses one resource of type PulseIn exclusively. The same resource index can only be used once in an applet.

For sending messages, at least ports PulseI, PulseEffectI, and PulseSyncRequestI need to be connected in a sensible way.

Depending on the type of the messages, also ports PulseColorSelectI, PulseFramePeriodeI, PulseStartIntRI, PulseStartIntGI, and PulseStartIntBI need to be connected sensibly. If these ports are not used, they nevertheless need to be connected. But in this case it doesn't matter what they are connected to.

For the individual CLHS message formats, you need sensible input at the following ports:

| Pulse Message Type | Direction | Ports |
| --- | --- | --- |
| 1 | Input | PulseI, PulseEffectI, PulseSyncRequestI |
| 2 / 3 | Input | PulseI, PulseEffectI, PulseSyncRequestI, PulseColorSelectI |
| 4 / 5 | Input | PulseI, PulseEffectI, PulseSyncRequestI, PulseFramePeriodeI, PulseStartIntRI |
| 6 / 7 | Input | PulseI, PulseEffectI, PulseSyncRequestI, PulseFramePeriodeI, PulseStartIntRI, PulseStartIntGI, PulseStartIntBI |

A pulse message is sent at each write to port PulseI. Thus, the parameters of the pulse message need to be valid when a pixel (0D) with the type of the pulse message is written to PulseI.

For each pixel incoming at port PulseI, a pulse message is generated. If the pixels are coming in too fast, messages are lost as a synchronization is required that needs a few clock cycles.

| Port Name | Direction | Type | Description |
|---|---|---|---|
| PulseI | Input | OM | Send a Pulse Message via CLHS (type is defined by the sent value) |
| PulseEffectI | Input | OM | Pulse Message Effect 7Bit, user defined |
| PulseSyncRequestI | Input | OM | Pulse Message Sync request. Next Image Package shall start at row = 0 with set MasterFGSync-Bit |
| PulseColorSelectI | Input | OM | Pulse Message Color select: 0 = all colors, 1 = Red, 2 = Green, 4 = Blue |
| PulseFramePeriodeI | Input | OM | Pulse Message Integration Period: Frame Period, load value for 32-bit counter. Condition 0xFFFFFFFF > Frame Period > Integration start > 0.<br><br>Frame Time = (frame Period + 3) * integration clock period |
| PulseStartIntRI | Input | OM | Pulse Message Integration start red, compare values for a 32 Bit down counter. Condition:<br><br>0xFFFFFFFF > Frame Period > Integration start > 0.<br><br>Integration time = integration start * integration clock period |
| PulseStartIntGI | Input | OM | Pulse Message Integration start green, compare values for a 32 Bit down counter. Condition:<br><br>0xFFFFFFFF > Frame Period > Integration start > 0.<br><br>Integration time = integration start * integration clock period |
| PulseStartIntBI | Input | OM | Pulse Message Integration start blue, compare values for a 32 Bit down counter. Condition:<br><br>0xFFFFFFFF > Frame Period > Integration start > 0.<br><br>Integration time = integration start * integration clock period |

---

**Supported Link Format**

| Link Parameter | PulseI | PulseEffectI | PulseSyncRequestI |
|---|---|---|---|
| Bit Width | 8 | 7 | 1 |
| Arithmetic | Unsigned | Unsigned | Unsigned |
| Parallelism | 1 | 1 | 1 |
| Kernel Columns | 1 | 1 | 1 |
| Kernel Rows | 1 | 1 | 1 |
| Image Protocol | VALT_PIXEL0D | VALT_PIXEL0D | VALT_PIXEL0D |
| Color Format | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| Color Flavor | FL_NONE | FL_NONE | FL_NONE |
| Maximal Image Width | 1 | 1 | 1 |
| Maximal Image Height | 1 | Any | 1 |

| Link Parameter | PulseColorSelectI | PulseFramePeriodeI | PulseStartIntRI | PulseStartIntGI | PulseStartIntBI |
|---|---|---|---|---|---|
| Bit Width | 8 | 32 | 32 | 32 | 32 |
| Arithmetic | Unsigned | Unsigned | Unsigned | Unsigned | Unsigned |
| Parallelism | 1 | 1 | 1 | 1 | 1 |
| Kernel Columns | 1 | 1 | 1 | 1 | 1 |
| Kernel Rows | 1 | 1 | 1 | 1 | 1 |
| Image Protocol | VALT_PIXEL0D | VALT_PIXEL0D | VALT_PIXEL0D | VALT_PIXEL0D | VALT_PIXEL0D |
| Color Format | VAF_GRAY | VAF_GRAY | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| Color Flavor | FL_NONE | FL_NONE | FL_NONE | FL_NONE | FL_NONE |
| Maximal Image Width | 1 | 1 | 1 | 1 | 1 |
| Maximal Image Height | 1 | 1 | 1 | 1 | 1 |

### 29.26.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | all Links, all Links |

### 29.26.2. Supported Link Format

| Link Parameter | Input Link all Links |
|---|---|
| **Bit Width** | see tables above |
| **Arithmetic** | see tables above |
| **Parallelism** | see tables above |
| **Kernel Columns** | see tables above |
| **Kernel Rows** | see tables above |
| **Img Protocol** | see tables above |
| **Color Format** | see tables above |
| **Color Flavor** | see tables above |
| **Max. Img Width** | see tables above |
| **Max. Img Height** | see tables above |

### 29.26.3. Parameters

None

# 29.27. Operator CLHSPulseOut

**Operator Library: Hardware Platform**

This operator manages the receiving of pulse messages.

| Available for Hardware Platforms |
| --- |
| microEnable 5 marathon VF2 |

The operator uses one resource of type PulseOut exclusively. The same resource index can only be used once in an applet.

For receiving messages, at least port PulseO needs to be connected.

Depending on the way you want to use the operator, further information of the pulse message can be received via ports PulseEffectO, PulseSyncRequestO, PulseColorSelectI, PulseFramePeriodeI, PulseStartIntRI, PulseStartIntGI, and PulseStartIntBI Depending on the type of the pulse messages, you need to connect specific out ports.

For the individual CLHS message formats, the following combinations are possible:

| Pulse Message Type | Direction | Ports |
| --- | --- | --- |
| 1 | Output | PulseO, PulseEffectO, PulseSyncRequestO |
| 2 / 3 | Output | PulseO, PulseEffectO, PulseSyncRequestO, PulseColorSelectO |
| 4 / 5 | Output | PulseO, PulseEffectO, PulseSyncRequestO, PulseFramePeriodeO, PulseStartIntRO |
| 6 / 7 | Output | PulseO, PulseEffectO, PulseSyncRequestO, PulseFramePeriodeO, PulseStartIntRO, PulseStartIntGO, PulseStartIntBO |

When a pulse message has been received, at out port PulseO a 0D pixel with a mode-of-message field is generated. The parameters of the pulse message are output on the according link o-synchroneously.

| Port Name | Direction | Type | Description |
|---|---|---|---|
| Pulse0 | Output | OM | Send a Pulse Message via CLHS (Type is defined by the send value) |
| PulseEffect0 | Output | OM | Pulse Message Effect 7Bit, user defined |
| PulseSyncRequest0 | Output | OM | Pulse Message Sync request. Next Image Package shall start at row = 0 with set MasterFGSync-Bit |
| PulseColorSelect0 | Output | OM | Pulse Message Color select: 0 = all colors, 1 = Red, 2 = Green, 4 = Blue |
| PulseFramePeriode0 | Output | OM | Pulse Message Integration Period: Frame Period, load value for 32-bit counter. Condition 0xFFFFFFFF > Frame Period > Integration start > 0.<br><br>Frame Time = (frame Period + 3) * integration clock period |
| PulseStartIntR0 | Output | OM | Pulse Message Integration start red, compare values for a 32 Bit down counter. Condition:<br><br>0xFFFFFFFF > Frame Period > Integration start > 0.<br><br>Integration time = integration start * integration clock period |
| PulseStartIntG0 | Output | OM | Pulse Message Integration start green, compare values for a 32 Bit down counter. Condition:<br><br>0xFFFFFFFF > Frame Period > Integration start > 0.<br><br>Integration time = integration start * integration clock period |
| PulseStartIntB0 | Output | OM | Pulse Message Integration start blue, compare values for a 32 Bit down counter. Condition:<br><br>0xFFFFFFFF > Frame Period > Integration start > 0.<br><br>Integration time = integration start * integration clock period |

---

**Supported Link Format**

| Link Parameter | PulseO | PulseEffectO | PulseSyncRequestO |
|---|---|---|---|
| Bit Width | 8 | 7 | 1 |
| Arithmetic | Unsigned | Unsigned | Unsigned |
| Parallelism | 1 | 1 | 1 |
| Kernel Columns | 1 | 1 | 1 |
| Kernel Rows | 1 | 1 | 1 |
| Image Protocol | VALT_PIXELOD | VALT_PIXELOD | VALT_PIXELOD |
| Color Format | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| Color Flavor | FL_NONE | FL_NONE | FL_NONE |
| Maximal Image Width | 1 | 1 | 1 |
| Maximal Image Height | 1 | Any | 1 |

| Link Parameter | PulseColorSelectO | PulseFramePeriodeO | PulseStartIntROO | PulseStartIntGO | PulseStartIntBO |
|---|---|---|---|---|---|
| Bit Width | 8 | 32 | 32 | 32 | 32 |
| Arithmetic | Unsigned | Unsigned | Unsigned | Unsigned | Unsigned |
| Parallelism | 1 | 1 | 1 | 1 | 1 |
| Kernel Columns | 1 | 1 | 1 | 1 | 1 |
| Kernel Rows | 1 | 1 | 1 | 1 | 1 |
| Image Protocol | VALT_PIXELOD | VALT_PIXELOD | VALT_PIXELOD | VALT_PIXELOD | VALT_PIXELOD |
| Color Format | VAF_GRAY | VAF_GRAY | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| Color Flavor | FL_NONE | FL_NONE | FL_NONE | FL_NONE | FL_NONE |
| Maximal Image Width | 1 | 1 | 1 | 1 | 1 |
| Maximal Image Height | 1 | 1 | 1 | 1 | 1 |

### 29.27.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | all Links, all Links |

### 29.27.2. Supported Link Format

| Link Parameter | Input Link all Links |
|---|---|
| **Bit Width** | see tables above |
| **Arithmetic** | |
| **Parallelism** | see tables above |
| **Kernel Columns** | see tables above |
| **Kernel Rows** | see tables above |
| **Img Protocol** | see tables above |
| **Color Format** | see tables above |
| **Color Flavor** | see tables above |
| **Max. Img Width** | see tables above |
| **Max. Img Height** | see tables above |

### 29.27.3. Parameters

None

# 29.28. Operator CLHSSingleCamera

**Operator Library: Hardware Platform**

This operator represents the image data interface between a CLHS Single Link configuration camera and VisualApplets. This operator receives data from the CLHS camera and feeds it into the image processing application.

| Available for Hardware Platform |
| --- |
| marathon VF2 |

The operator uses one resource of type camera exclusively. You can select port 0 or port 1 of the frame grabber as image source. The same resource ID cannot be used more than once in the applet.

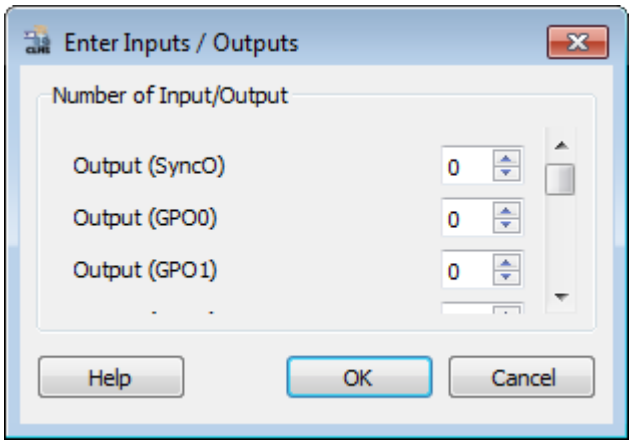
This operator has always one output port O, which is the image output.

**Adding Optional GPIO Ports**

In addition to this standard output port, you can specify various optional ports if required by your design.

The CLHS protocol supports pulses and GPIO messages.

During instantiation of the operator, a pop-up dialog appears:




Here, you can define the availability of up to 16 optional GPIs (General Purpose Inputs) and up to 16 optional GPOs (General Purpose Outputs).

If you set the port availability to "0" (default), the port will not be present in the operator. If you set the port availability to 1, the particular port is available at the operator interface.

The GPIO state is transferred when the state of a GPI is changed by a connected operator. When the operator receives a GPIO message from the camera, all GPOs are updated.

**Parameter Bit Width**

Link parameter Bit Width can be set to a value higher or lower than the bit width the camera is actually sending:

- If you select a lower value, the LSB bits of each pixel are cut off, so that only the MSB bits are transferred into the application.
- If you select a higher value, the original pixel is set to the MSB position of the outgoing pixel. The LSB bits are filled with zeros.

In both cases, the relative brightness remains the same, i.e., white pixels remain white and dark pixels remain dark.

### Adding Optional Output Ports

In addition, you can (optionally) activate two additional output ports:

- one output port **Output (SyncO)** (MultiFGSync bit of video message), and
- one output port **Output (HeaderO)**. **Output (HeaderO)** provides the header information of the video package that is currently worked on. The port provides the header information in a pixel of a width of 64 bit.

### The Structure of Output Port HeaderO

Before the image data are output on output port O, one single pixel is output on port **HeaderO**. This pixel contains the data of the header of the video package. Output port **HeaderO** is used to transfer information about more complex dependencies of the incoming data, or to react to events via the video link of the applet.

The structure of output port **HeaderO** is as follows:

| Position | CLHS Header | Description |
|---|---|---|
| [0,2] | ColorID [0,2] | Video Source: At the moment this part is always 0 |
| [3,4] | ColorID [3,4] | Packet Type 0 = video (all other combinations are reserved) |
| [5,7] | ColorID [5,7] | Color ID: |

| ColorID | Description |
|---|---|
| 0 | Mono, Multi-component, Bayer, Raw |
| 1 | Blue |
| 2 | Green |
| 3 | Red |
| 4 | Alpha |
| (5..7) | reserved |

Table 29.6.

| | | |
|---|---|---|
| [8,9] | VideoStatus [1,0] | Row Marker: |

| Bits | Description |
|---|---|
| 00 | Continuation of Row |
| 01 | Start of Row |
| 10 | End of Row |
| 11 | Complete row (start & end) |

Table 29.7.

| | | |
|---|---|---|
| [10,11] | VideoStatus [3,2] | Frame Marker: |

| Bits | Description |
|---|---|
| 00 | Continuation of Frame |
| 01 | Start of Frame |

| | | |
|---|---|---|
| | | 10 End of Frame<br>11 Complete Frame (start & end) |

Table 29.8.

| | | |
|---|---|---|
| 12 | VideoStatus [4] | Missed Trigger. Camera was not able to react to a trigger request since last package |
| 13 | VideoStatus [5] | Buffer overflow in Camera |
| 14 | VideoStatus [6] | MultiFGSync Synchronisation for multi FG usage(see also SyncO output) |
| 15 | VideoStatus [7] | Resend Flag, set to 0, not used |
| [16,31] | RowID | Number of the current Row |
| [32,39] | - | Reserved, set to 0 |
| [40,55] | ColID | Number of the Column of the first pixel in this package |
| [56,63] | AcquisitionSet | This Byte may be used by the camera to set proprietary information |

Table 29.5.

### 29.28.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Links** | O, Acquisition image data to be used inside VisualApplets<br>GPO[0;15] (optional), Status of the Camera GPOs (Exchange via CLHS)<br>GPI[0;15] (optional), Status of the Framegrabber GPOs (Exchange via CLHS)<br>SyncO (optional), MultiFGSync Bit is set in Video Package (valid for 1 clock cycle)<br>HeaderO (optional), One 64 bit wide Pixel at the start of each video package containing the video package header |

### 29.28.2. Supported Link Format

| Link Parameter | Output Link O | Output Link GPO[0;15] (optional) | Output Link GPI[0;15] (optional) |
|---|---|---|---|
| **Bit Width** | any | 1 | 1 |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | any, see parameter description | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |

| Link Parameter | Output Link O | Output Link GPO[0;15] (optional) | Output Link GPI[0;15] (optional) |
|---|---|---|---|
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | any | 1 | 1 |
| **Max. Img Height** | any | 1 | 1 |

| Link Parameter | Output Link SyncO (optional) | Output Link HeaderO (optional) |
|---|---|---|
| **Bit Width** | 1 | 64 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_Pixel0D |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | 1 | 1 |
| **Max. Img Height** | 1 | 1 |

### 29.28.3. Parameters

| **CameraID** | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | {0, 1} |
| The parameter specifies which camera resource is used. Furthermore, the ID will be used to map camera channels to applet ports. | |

| **PixelFormat** | |
|---|---|
| **Type** | static write parameter |
| **Default** | Mono10p |
| **Range** | {Mono8,Mono10p,Mono12p,Mono14p,Mono16,Raw8,Raw10p,Raw12p,Raw14p,Raw16, Raw64, B8, B10p, B12p, B14p, B16, G8, G10p, G12p, G14p, G16, R8, R10p, R12p, R14p, R16, BGR8, BGR10p, BGR12p, BGR14p, BGR16, BGRa8, BGRa10p, BGRa12p, BGRa14p, BGRa16, BayerGR8, BayerGR10p, BayerGR12p,BayerGR14p, BayerGR16, BayerRG8, BayerRG10p, BayerRG12p, BayerRG14p, BayerRG16, BayerGB8, BayerGB10p, BayerGB12p, BayerGB14p, BayerGB16, BayerBG8, BayerBG10p, BayerBG12p, BayerBG14p, BayerBG16} |
| The parameter specifies the output pixel format.<br><br>In RAW mode, all line and frame markers coming in from the camera will be ignored. RAW mode is used to support complex link structures and is exclusively available in operator CLHSSingleCamera. | |

| **AquisitionFormat** | |
|---|---|
| **Type** | static write parameter |
| **Default** | Area |
| **Range** | {Area; Linescan; RAW} |
| When this parameter is set to "Area", the first data available is a complete frame. Frames started in the middle will be ignored. Therefore, a "Start Of Frame" marker in the first video message is required. | |

| **AquisitionFormat** |
|---|
| When this parameter is set to "Line", the first data available is a complete line. Lines started in the middle will be ignored. Therefore, a "Start Of Line" marker in the first video message is required. "Start of Frame" and "End of Frame" will be ignored, since they are set to 0 for linescan cameras.<br><br>When this parameter is set to "RAW", all markers will be ignored ("Start/End of Line" as well as "Start/End of Frame"). "RAW" mode is used to support complex link structures. This mode is only available in the *CLHSSingleCamera* operator. |

| **LineSupervision** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | off |
| **Range** | {on, off} |
| If this parameter is set to "on", the operator monitors if all lines incoming from the camera are transferred to the applet.<br><br>The lines comming in from the camera are numbered in ascending order (line 0, line 1, line 2 ... line N). The parameter makes sure that no lines are lost. For example, if a camera with one link is connected and the parameter monitors a sequence like 332, 333, **335**, 336 …, an error is thrown.<br><br>If a camera has more than 1 link the step size for incoming lines per link can be increased using parameter *RowStep* accordingly. For example, if a camera has 2 links, link 0 receives lines 0, 2, 4, 6, … , and link 1 receives lines 1, 3, 5, 7 … etc. Accordingly, if a camera has 4 links, link 0 receives lines 0, 4, 8, 12, … , link 1 receives lines 1, 5, 9, 13, link 2 receives lines 2, 6, 10, 14, … etc.<br><br>Furthermore, parameter *LineSupervision* checks for integrity: Start of line (SoL) and End of Line (EoL) must be in the correct order. Missing EoL or SoL will be detected, an error flag is generated and an End of Line is inserted. Missing Lines will be inserted as smaller lines (length depends on parallelism and bitwidth) until 2 following RowIDs are detected and line consistency is guaranteed again. | |

| **RowStep** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | {1, 65535} |
| Step between 2 RowIDs. Should be 1 unless a multi-channel camera is used and lines are split amongst links. | |

| **SyncFirstLineToFGSyncRequest** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | no |
| **Range** | {no,yes} |
| If this parameter is set to yes, all lines before the first FGSyncRequest are discarded. Works only in LINE_1D mode. This synchronizes the first line with the first external FGSyncRequest pulse. | |

| **MinimalParallelism** | |
|---|---|
| **Type** | static read parameter |
| **Default** | 2 |
| **Range** | [1; 1024] |
| Minimal parallelism for the output link O to be able to transport the maximal bandwidth of the camera without losing data. This value depends on the currently selected design clock frequency. The higher the frequency the lower the parallelism value can become. | |

# 29.29. Operator CxpCamera

**Operator Library: Hardware Platform**

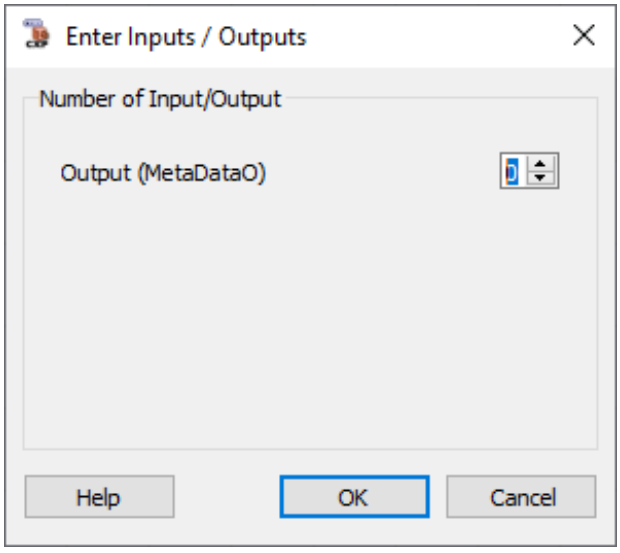
This operator represents the image data interface between a CXP camera and VisualApplets. The operator can be used for single channel, dual channel or quad channel CXP interfaces. The type of interface is selected by the parameter *ConnectionCount*. The operator outputs raw image data no matter what format the camera delivers. You then need to convert this image data into the format sent by the camera. To convert the image data, use appropriate operators for aggregating and casting raw byte values to pixel values . Certain situations during operation may be communicated via the event system as described below. Additionally, the operator has various parameters signalling the status of the connection to the camera.

If the operator detects that a frame is larger or smaller than what was promoted by the camera in the CXP image header, a safety circuit gets activated. The operator will then cut off exceeding pixels and lines, so that VisualApplets sees always the frame size which was defined in the image header. If the frame is smaller in its dimensions than what was specified in the image header, the operator will fill 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 following frames to get lost. The loss will be reported per event to the software, also the size mismatch will cause an event, too.

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |

**Instantiation in VisualApplets**

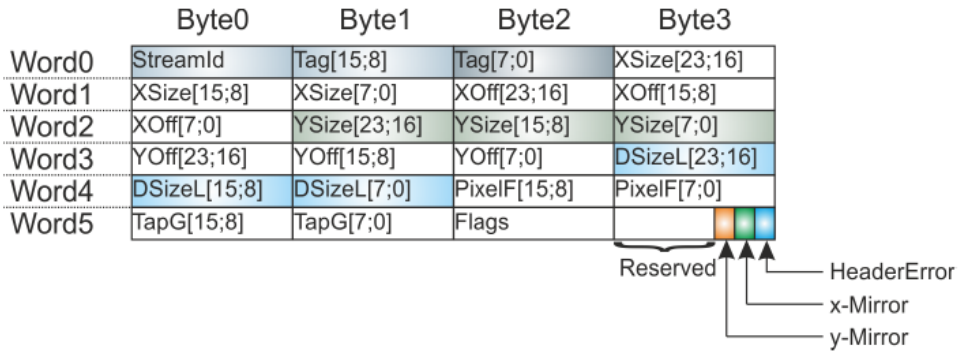
The operator provides image data on its output O. This output is always present. In addition to this standard output port, you can configure an optional *MetaDataO* output for the CXP header meta-data. The following pop-up dialog appears during operator instantiation:




Here, you can specify the optional meta data port. If you set the port availability to "0" (default), the port will not be present in the operator. If you set the port availability to 1, the meta data port is available in the operator interface.

**Optional MetaDataO Port**

The format for the 32-bit output is shown in the following picture.

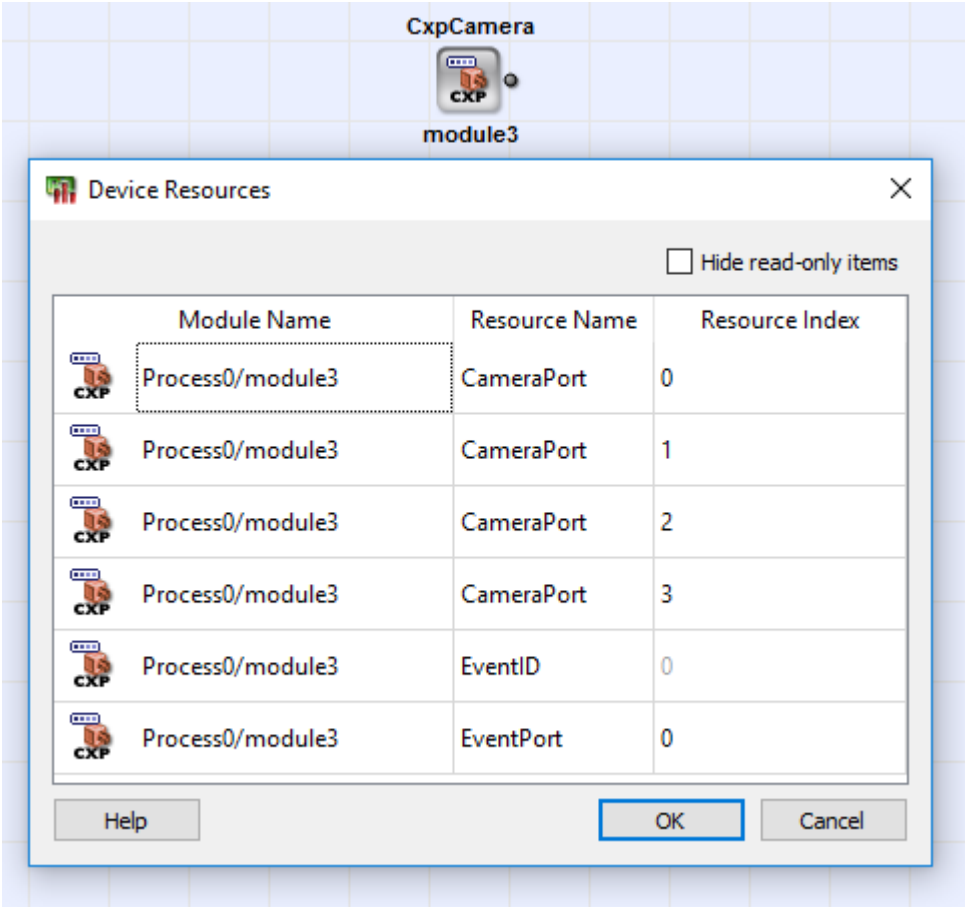



Each CXP frame provides a corresponding error-free image header. For incorrect image header, no image stream data is sourced into the VisualApplets pipeline. The compressed image header consist of 6 words. The last byte is used for additional information, which is for internal usage only. Particularly, the HeaderError bit informs if the image header itself got an error or not.

| Imageeader | Description |
|---|---|
| **StreamId** | ID of the CXP stream |
| **Tag** | 16-bit source image index. It is incremented for each transferred image, wraps around to 0 at 0xFFFF. The same number shall be used by each stream containing data relating to the same image (in case of multi-tap streams). |
| **XSize** | 24-bit value representing the image width in pixels. |
| **XOff** | 24-bit value representing the horizontal offset in pixels of the image with respect to the left hand pixel of the full device image. |
| **YSize** | 24-bit value representing the image height in pixels. This value is set to 0 for line scan images. |
| **YOff** | 24-bit value representing the vertical offset in pixels of the image with respect to the top line of the full device image. This value is set to 0 for line scan images. |
| **DSizeL** | 24-bit value representing the number of data words per image line. |
| **PixelF** | 16-bit value representing the pixel format. |
| **TapG** | 16 bit value representing the tap geometry. |
| **Flags** | Image flags. |
| **x-Mirror** | (Not used yet). Describes whether the incoming image is x-mirrored or not. |
| **y-Mirror** | (Not used yet). Describes whether the incoming image is y-mirrored or not. |
| **HeaderError** | 1: Image header has an error and the frame is declared as lost. There is no corresponding image data exists in the operator output data stream.<br><br>0: Image header is correct and there is a corresponding image data exists in the operator output data stream. |

**Device Resource Usage**

The operator uses one or more resources of type *CameraPort* depending on the selection of the parameter *ConnectionCount*. For the event *system* a resource of the type *EventPort* is used. If *ConnectionCount* is set to *X4* (quad channel), the following resource dialog opens up:



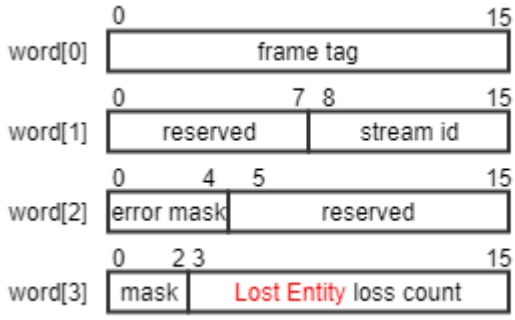

**Modifying Image Width and Image Height**

Via the maximum image width and height properties of the output link you can adjust width and height to the camera-specific settings. However, the maximum image width on operator port O **must be divisible by the parallelism** of port O. Thus, make sure the maximum image width is divisible by the parallelism of port O!

**Error Handling and Event System**

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 of a smaller frame can cause the follow-up frames to get lost. The loss is then reported per event to the software (see the following paragraph). The size mismatch causes an event, too.

For a set of very critical errors, the operator will forward asynchronous events to the host runtime software (Framegrabber SDK). The event name in the Framegrabber API is <hierarchical operator name>\CxpStreamStatus, e.g. Device1\Process0\Camera\CxpStreamStatus. 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 is not 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 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 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]: 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]: Event loss for type **Lost Entity** occurred. This means that preceding events of type **Lost Entity** got lost. This happens when the runtime software 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 happens. 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, when a CRC error is detected after the preceding frame was already sent by the operator. In normal situations, where 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.

### 29.29.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Links** | O, image data output<br>MetaDataO, optional meta data output |

### 29.29.2. Supported Link Format

| Link Parameter | Output Link O | Output Link MetaDataO |
|---|---|---|
| **Bit Width** | | 32 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | auto | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} (default: VALT_IMAGE2D) | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} (default: VALT_IMAGE2D) |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any (default: 1032) | 6 |
| **Max. Img Height** | any (default: 1032) | 1 |

### 29.29.3. Parameters

| ConnectionCount | |
|---|---|
| **Type** | Static Write parameter |
| **Default** | X1 |
| **Range** | {X1, X2, X4} |

| **ConnectionCount** |
|---|
| The parameter *ConnectionCount* defines the number of CXP lanes aggregated to the CXP link. The indices of the used connection ports are handled by resources of the type *CameraPort*. The number of port resources matches the connection count (e.g. **X2**: two *CameraPort* resource items).<br><br>When you instantiate more than one *CxpCamera* operator (e.g. for dual-camera applet), resource conflicts may occur when multiple resources with the same index are used or when the number of consumed *CameraPort* resources exceeds the maximum of 4. In this case, the design rules check reports an error. |

| **ResetStatus** | |
|---|---|
| **Type** | Dynamic Write parameter |
| **Default** | Off |
| **Range** | {Off, On} |
| The parameter *ResetStatus* resets the camera statistics, i.e. the error counters. | |

| **UsedConnections** | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | {1,2,4} |
| The parameter *UsedConnections* shows the amount of CXP lanes configured by the discovery software at runtime. | |

| **PacketTagErrorCount** | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |
| The parameter *PacketTagErrorCount* shows how many received packets have a tag that is non-compliant with the expected tag according to the CXP standard. In particular this value is counting up when gaps are observed in following stream packet tag enumerations. 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. | |

| **ImageTagErrorCount** | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |
| The parameter counts how many mismatches occur between the image header tag, the expected tag according to the CXP standard and the received tag. 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. | |

| **StreamIdErrorCount** | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |
| 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. | |

| **CorrectedErrorCount** | |
|---|---|
| **Type** | Dynamic Read parameter |

| CorrectedErrorCount | |
|---|---|
| **Default** | |
| **Range** | [0 : 8191] |
| The parameter counts how many detected errors in the image header or the line markers have been corrected. 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. | |

| UncorrectedErrorCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |
| The parameter counts how many detected errors in the image header or the line markers could not be corrected due to multiple bit errors in same byte. 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. | |

| PacketBufferOverflowCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |
| The parameter counts how often the packet buffer overflow occurs in the channel bonding in aggregated mode. This parameter is only relevant for ConnectionCount = **X2** or **X4**. 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. | |

| PacketBufferOverflowSource | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0x0 : 0xf] |
| The parameter implements a bit mask to query in which of the potential 4 CXP channels the packet buffer overflow occurred. The parameter width depends on the parameter *ConnectionCount*. In **X1** mode, the parameter width is 1 bit wide, in **X2** mode, the parameter width is 2 bit wide and in **X4** mode the parameter width is 4 bit wide. The order is: LSB = lowest CXP channel number, MSB = highest CXP channel number allocated by the operator. | |

| CameraScanMode | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | {area,line} |
| The received image header carries the information whether the stream is for the area scan or for the line scan applications. This parameter shows the last valid received stream image header information. | |

| MarkerErrorCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |
| The parameter counts how often the sequence of the CXP stream marker and the header or the line markers was 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. | |

| UnexpectedStartupData | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | {false, true} |
| The parameter detects the error situation in which the first data value after the operator reset was unexpected, i.e. no image header is received before. 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. | |

| FrameLostCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 33554431] |
| The parameter counts the frames that were lost during acquisition and are not sent into the VisualApplets 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 the bit [24] stands for the counter overflow. When the overflow bit is set, the counter value shall be treated as don't care. | |

| FrameCorruptedCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 33554431] |
| The parameter counts the corrupted frames during acquisition. Corrupted frames are frames with error pixels which are sent to the VisualApplets pipeline. The parameter is 25 bit wide where the bits [23:0] represent the actual counter value and the bit [24] stands for the counter overflow. When the overflow bit is set, the counter value shall be treated as don't care. | |

### 29.29.4. Examples of Use

The use of operator CxpCamera is shown in the following examples:

- Section 10.5.1, 'CoaXPress Area Scan Cameras'

  Tutorial - Basic Acquisition

- Section 10.5.2, 'CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 29.30. Operator CxpCameraMultiTap

**Operator Library: Hardware Platform**

This operator represents the image data interface between a CXP dual-tap camera and VisualApplets. You can use the operator for single channel, dual channel or quad channel CXP interfaces. You select the type of interface with the parameter *ConnectionCount*. The operator outputs raw image data for each tap port, no matter which format the camera delivers. You then need to convert this image data into the format sent by the camera. To convert the image data, use appropriate operators for aggregating and casting raw byte values to pixel values. Certain situations during operation may be communicated via the event system as described below. Additionally, the operator has various parameters signaling the status of the connection to the camera.

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |

**CXP Standard Multi-Tap Definition**

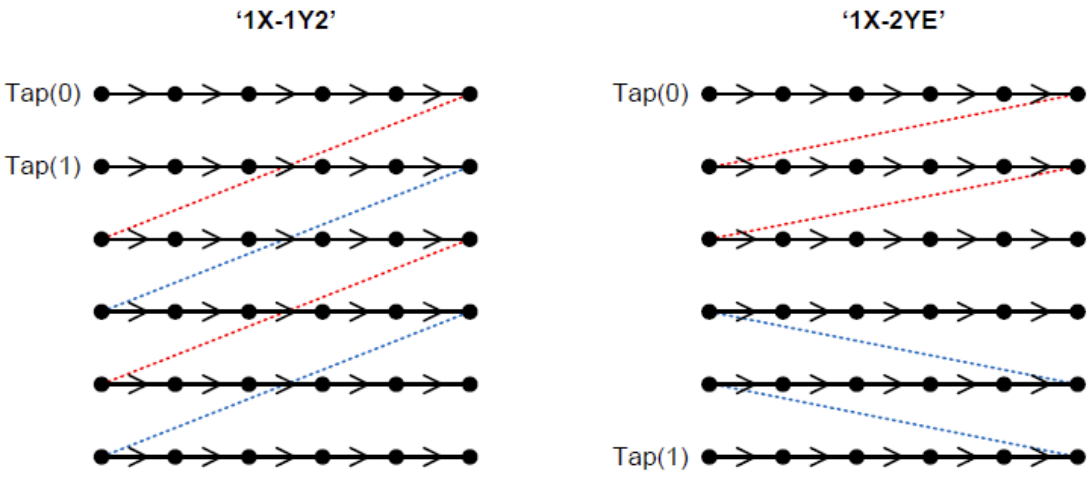
Usually, the image pixels are scanned sequentially from the top left to the bottom right. However, with the steadily growing number of pixels on modern image sensors, this approach does not produce acceptable frame rates. To tackle this issue, sensor manufacturers help themselves by scanning multiple pixels at the same time. The CXP standard accommodates this technique by introducing the concept of **taps**. A tap can be seen as a scanning device, which reads the image pixels sequentially. The way an image is scanned is defined as the *Tap Geometry*, which the host reads from the device.



**Taps Apply to Vertical Scanning Only**

Taps apply to vertical scanning only. Horizontal scanning is fixed from left to right.

It is recommended in the CXP standard, that a frame grabber host supports the three tap formats *1X-1Y*, *1X-1Y2*, and *1X-2YE*. The tap format *1X-1Y* represents the default characteristic scanning the image pixel by pixel from top left to bottom right, while the other formats represent two tap geometries, that are depicted in the following figure:




Although data from each tap is formed into separate streams, it is important to note that there is no fixed mapping between the stream-ID and the tap. However, it is safe to assume, that this mapping remains constant during acquisition. The TapG code, and thus, the required information to find this mapping, is provided in the CXP image header, which can be read on the optional *MetaDataTap0/1* ports.

| Tap Format | TapG Code |
|---|---|
| 1X-1Y | 0x0000 |

| Tap Format | TapG Code |
|---|---|
| 1X-1Y2, tap 1 | 0x0004 |
| 1X-1Y2, tap 2 | 0x1004 |
| 1X-2YE, tap 1 | 0x0041 |
| 1X-2YE, tap 2 | 0x1041 |

**Flipping the Lower Half of the Image Is Not the Responsibility of this Operator**

Flipping the lower half of the image in the case of the *1X-2YE* tap format is not the responsibility of this *CxpCameraMultiTap* operator. The user application needs to reformat the tap(1) by using additional VisualApplets operators.

### Instantiation in VisualApplets

The operator provides image data on its output tap. This output is always present. It is 1 for the single tap and needs to be set to 2 for the dual-tap cameras. In addition to this standard output ports, you can configure an optional MetaData output for the CXP header metadata for each tap output exclusively, i.e. for dual-tap application you can define up to 2 *MetaDataTap* ports. The following pop-up dialog appears during operator instantiation and can be configured to the following permutations:

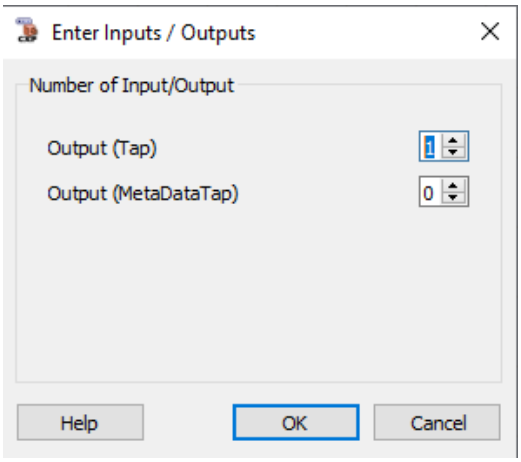



Figure 29.1. This configuration is equivalent to a simpler *CxpCamera* operator with only O output and no meta data.

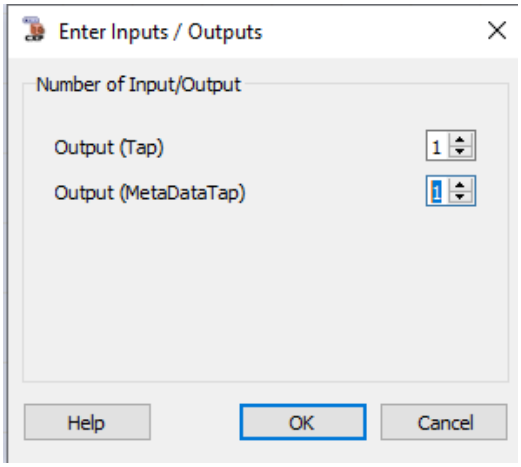



Figure 29.2. This configuration is equivalent to a simpler *CxpCamera* operator with port O and *MetaDataO* selected.

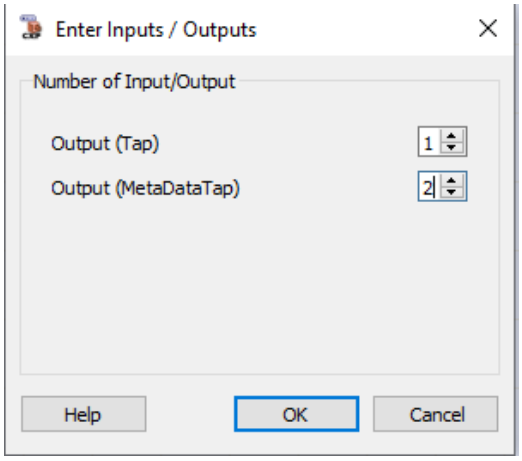



Figure 29.3. In this configuration only the tap 0 is output with its metadata. For the tap 1 only the metadata is output. This configuration can be useful for debugging a camera/frame grabber combination.

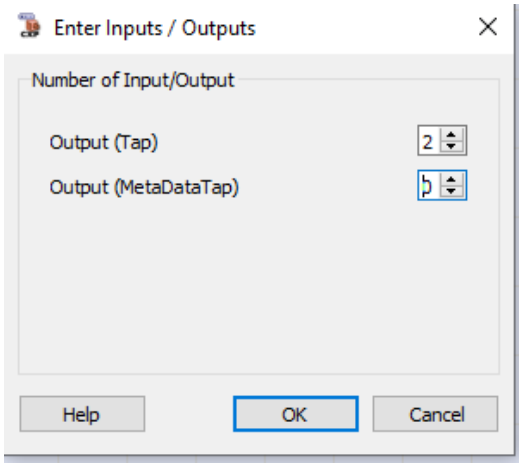



Figure 29.4. In this configuration the operator provides both camera taps as 2 separate data streams on its *Tap0* and *Tap1* ports. However, no metadata information is output.

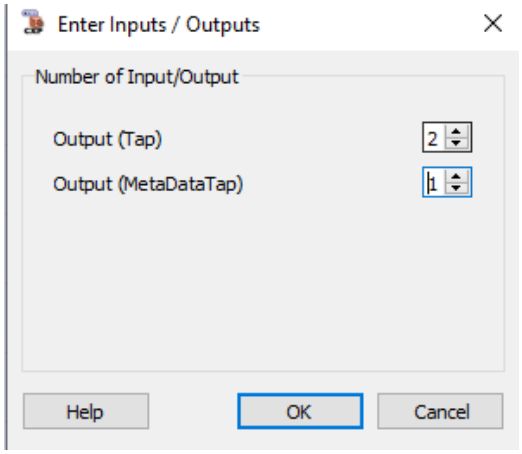



Figure 29.5. In this configuration the operator provides both camera taps as 2 separate streams on its *Tap0* and *Tap1* ports. The operator provides also the meta information for the *Tap0* port. This configuration might be meaningful for symmetrical camera tap configurations, where the CXP header is in most parts identical for both taps except for the *TapG Code* fields.

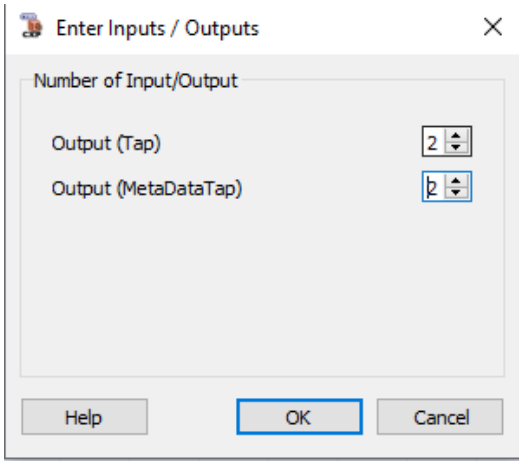
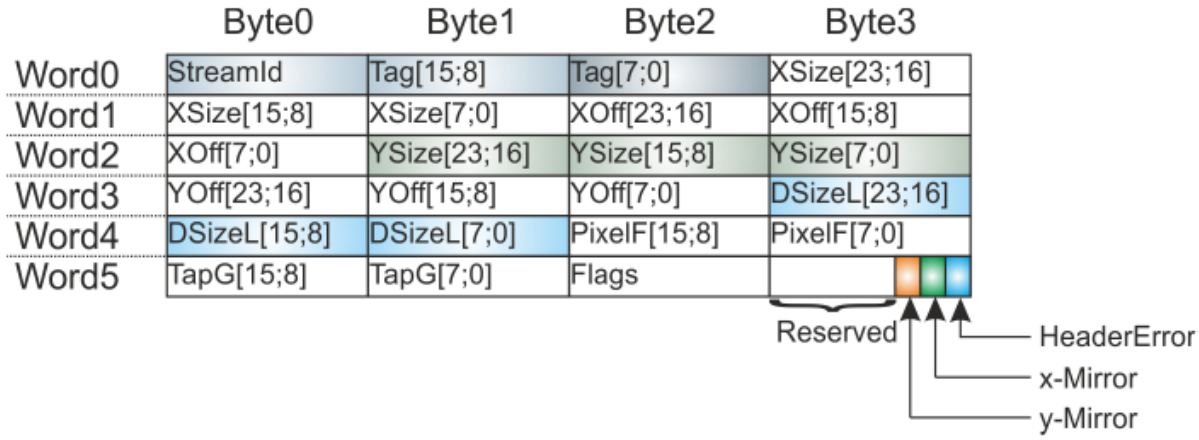



Figure 29.6. This is the maximal configuration of the operator, where for each tap an own output is presented together with its own metadata.

**Optional MetaDataTap Port**

The format for both metadata ports is identical. It is provided in VisualApplets as 32-bit output and is shown in the following picture:




Each CXP frame provides a corresponding error-free image header. For incorrect image headers, no image stream data is sourced into the VisualApplets pipeline. The compressed image header consist of 6 words. The last byte is used for additional information, which is for internal usage only. Particularly, the HeaderError bit informs whether the image header itself has an error or not.

| Image Header | Description |
|---|---|
| **StreamId** | ID of the CXP stream |
| **Tag** | 16-bit source image index. It is incremented for each transferred image, wraps around to 0 at 0xFFFF. The same number shall be used by each stream containing data relating to the same image (in case of multi-tap streams). |
| **XSize** | 24-bit value representing the image width in pixels. |
| **XOff** | 24-bit value representing the horizontal offset in pixels of the image with respect to the left hand pixel of the full device image. |
| **YSize** | 24-bit value representing the image height in pixels. This value is set to 0 for line scan images. |
| **YOff** | 24-bit value representing the vertical offset in pixels of the image with respect to the top line of the full device image. This value is set to 0 for line scan images. |
| **DSizeL** | 24-bit value representing the number of data words per image line. |

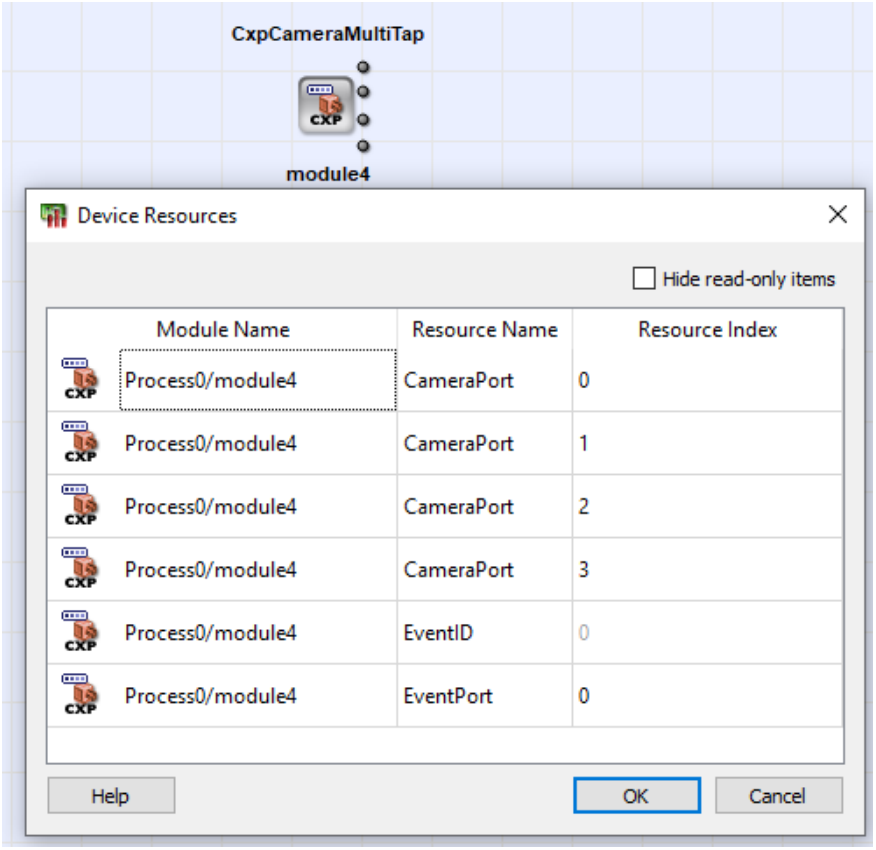
| Image Header | Description |
|---|---|
| **PixelF** | 16-bit value representing the pixel format. |
| **TapG** | 16 bit value representing the tap geometry. |
| **Flags** | Image flags. |
| **x-Mirror** | (Not used yet). Describes whether the incoming image is x-mirrored or not. |
| **y-Mirror** | (Not used yet). Describes whether the incoming image is y-mirrored or not. |
| **HeaderError** | 1: Image header has an error and the frame is declared as lost. No corresponding image data exists in the operator output data stream.<br><br>0: Image header is correct and a corresponding image data exists in the operator output data stream. |

**Modifying Image Width and Image Height**

Via the maximum image width and height properties of the output link you can adjust width and height to the camera-specific settings. However, the maximum image width on operator port O **must be divisible by the parallelism** of port O. Thus, make sure the maximum image width is divisible by the parallelism of port O!

### Device Resource Usage

The operator uses one or more resources of type *CameraPort* depending on the selection of the parameter *ConnectionCount*. For the event *system* a resource of the type *EventPort* is used. If *ConnectionCount* is set to *X4* (quad channel), the following resource dialog opens up:

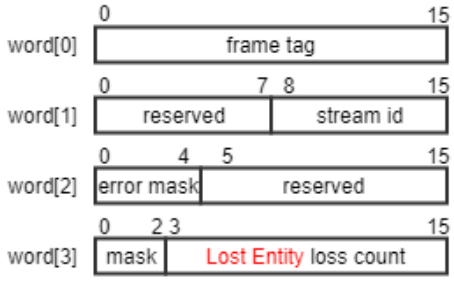



### Error Handling and Event System

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.

For a set of very critical errors, the operator will forward asynchronous events to the host runtime software (Framegrabber SDK). The event name in the Framegrabber API is <hierarchical operator name>\CxpStreamStatus, e.g. Device1\Process0\Camera\CxpStreamStatus. 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 is not 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 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.

### 29.30.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Links** | Tap0/Tap1, image data output MetaDataTap0/MetaDataTap1, optional meta data output |

### 29.30.2. Supported Link Format

| Link Parameter | Output Link Tap0/Tap1 | Output Link MetaDataTap0/ MetaDataTap1 |
|---|---|---|
| **Bit Width** | 8 | 32 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | auto | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |

| Link Parameter | Output Link Tap0/Tap1 | Output Link MetaDataTap0/ MetaDataTap1 |
|---|---|---|
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} (default: VALT_IMAGE2D) | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} (default: VALT_IMAGE2D) |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any (default: 1032) | 6 |
| **Max. Img Height** | any (default: 1032) | 1 |

### 29.30.3. Parameters

| ConnectionCount | |
|---|---|
| **Type** | Static Write parameter |
| **Default** | X1 |
| **Range** | {X1, X2, X4} |

The parameter *ConnectionCount* defines the number of CXP lanes aggregated to the CXP link. The indices of the used connection ports are handled by resources of the type *CameraPort*. The number of port resources matches the connection count (e.g. **X2**: two *CameraPort* resource items).

When you instantiate more than one *CxpCamera* operator (e.g. for dual-camera applet), resource conflicts may occur when multiple resources with the same index are used or when the number of consumed *CameraPort* resources exceeds the maximum of 4. In this case, the design rules check reports an error.



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *ConnectionCount* parameter counts the used connection ports across both tap streams.

| ResetStatus | |
|---|---|
| **Type** | Dynamic Write parameter |
| **Default** | Off |
| **Range** | {Off, On} |

The parameter *ResetStatus* resets the camera statistics, i.e. the error counters.



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *ResetStatus* parameter resets the camera statistics across both tap streams.

| UsedConnections | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | {1,2,4} |

The parameter *UsedConnections* shows the amount of CXP lanes configured by the discovery software at runtime.

| UsedConnections |
|---|

**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *UsedConnections* parameter represents the configured ports across both tap streams.

| PacketTagErrorCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |

The parameter *PacketTagErrorCount* shows how many received packets have a tag that is non-compliant with the expected tag according to the CXP standard. In particular, this value is counting up when gaps are observed in following stream packet tag enumerations. 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.

**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *PacketTagErrorCount* parameter counts the packets with not expected tags across both tap streams.

| ImageTagErrorCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |

The parameter counts how many mismatches occur between the image header tag, the expected tag according to the CXP standard and the received tag. 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.

**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *ImageTagErrorCount* parameter counts the mismatches of tags across both tap streams.

| StreamIdErrorCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |

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.

**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *StreamIdErrorCount* parameter counts the stream-IDs across both tap streams.

| CorrectedErrorCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |

| CorrectedErrorCount | |
|---|---|
| **Range** | [0 : 8191] |

The parameter counts how many detected errors in the image header or the line markers have been corrected. 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.

| UncorrectedErrorCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |

The parameter counts how many detected errors in the image header or the line markers could not be corrected due to multiple bit errors in same byte. 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.



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the error monitoring parameter represents the sum of the corresponding error types across both tap streams.

| PacketBufferOverflowCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |

The parameter counts how often the packet buffer overflow occurs in the channel bonding in aggregated mode. This parameter is only relevant for ConnectionCount = **X2** or **X4**. 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.



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *PacketBufferOverflowCount* parameter counts the packet buffer overflows across both tap streams.

| PacketBufferOverflowSource | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0x0 : 0xf] |

The parameter implements a bit mask to query in which of the potential 4 CXP channels the packet buffer overflow occurred. The parameter width depends on the parameter *ConnectionCount*. In **X1** mode, the parameter width is 1 bit wide, in **X2** mode, the parameter width is 2 bit wide and in **X4** mode the parameter width is 4 bit wide. The order is: LSB = lowest CXP channel number, MSB = highest CXP channel number allocated by the operator.



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *PacketBufferOverflowSource* parameter searches for overflows across both tap streams.

| CameraScanMode | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |

| CameraScanMode | |
|---|---|
| **Range** | {area,line} |

The received image header carries the information whether the stream is for the area scan or for the line scan applications. This parameter shows the last valid received stream image header information.

This parameter is read out only for the Tap0 stream with the assumption, that Tap1 stream is exactly the same mode. This means this parameter applies either to area scan or to line scan for the complete camera and is thus identical for both taps and is not tap-specific.

| MarkerErrorCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 8191] |

The parameter counts how often the sequence of the CXP stream marker and the header or the line markers was 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.



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the error monitoring parameter represents the sum of the corresponding error types across both tap streams.

| UnexpectedStartupData | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | {false, true} |

The parameter detects the error situation in which the first data value after the operator reset was unexpected, i.e. no image header is received before. 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.



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the error monitoring parameter represents the sum of the corresponding error types across both tap streams.

| FrameLostCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 33554431] |

The parameter counts the frames that were lost during acquisition and are not sent into the VisualApplets 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 the bit [24] stands for the counter overflow. When the overflow bit is set, the counter value shall be treated as don't care.



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *FrameLostCount* parameter counts the lost frames across both tap streams.

| FrameCorruptedCount | |
|---|---|
| **Type** | Dynamic Read parameter |
| **Default** | |
| **Range** | [0 : 33554431] |

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



**All Parameters Are Common to Both Taps**

All parameters are common to both taps, i.e. the *FrameCorruptedCount* parameter counts the corrupted frames across both tap streams.

## 29.30.4. Examples of Use

The use of operator CxpCameraMultiTap is shown in the following examples:

- Section 11.5, 'Functional Example for Multi Tap Camera Interface with Tap Geometry Sorting '

  Examples - Demonstration of how to use the operator

# 29.31. Operator CxpAcquisitionStatus

**Operator Library: Hardware Platform**

The operator *CxpAcquisitionStatus* provides the acquisition status as reported by the runtime software.

The operator is using one resource of type *AcquisitionCameraPort* which is in range 0 to 3 and corresponds to the CXP channel. The *AcquisitionCameraPort* resource corresponds to the *CameraPort* resource which is used by the operator *CxpCamera*.

The output of this operator is obtained by a firmware register controlled by the runtime software. The output changes whenever the runtime software changes the CXP streaming status between *running* and *stopped*.

You may use the output of this operator to gate a circuit for trigger signal generation, so the trigger pulses are only generated when streaming is ongoing.

## 29.31.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, status output |

## 29.31.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | 16777216 |
| **Max. Img Height** | 16777216 |

## 29.31.3. Parameters

# 29.32. Operator CxpPortStatus

**Operator Library: Hardware Platform**

The operator monitors the status of a CXP channel. It displays the current channel configuration and provides access to error counters via read parameters.

The operator is using one resource of type *CxpStatusPort* which is in range 0 to 3 for imaFlex CXP-12 Quad, and in range 0 to 4 for imaFlex CXP-12 Penta. This resource corresponds to the CXP channel. The *CxpStatusPort* resources correspond to the *CameraPort* resources which are used by operator *CxpCamera*.

| Available for Hardware Platform |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |

## 29.32.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | None (since there are neither Inputs nor Outputs) |

## 29.32.2. Supported Link Format

None

## 29.32.3. Parameters

| PoCXPState | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | BOOTING |
| **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 ADC chip is broken, which measures the voltage and current of the channel. | |

| PoCXPCurrent | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0:1000] |
| Current flowing through the CXP cable in mA. | |

| PoCXPVoltage | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 30] |
| Voltage sourced through the CXP cable in V. | |

| PoCXPControllerEnabled | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | YES |

| PoCXPControllerEnabled | |
|---|---|
| **Range** | {NO,YES} |
| State of the power over CXP engine. YES means the power controller will source the camera which requires power over CXP cable when connected, NO means the camera is not powered via the CXP cable. Note that this parameter is not showing whether the camera is sourced or not, instead it is showing that the capability of powering the camera via the CXP cable is enabled or not. | |

| MappedToFgPort | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | imaFlex CXP-12 Quad:[0:3]<br>imaFlex CXP-12 Penta:[0:4] |
| Frame grabber port to which the corresponding CXP channel is connected. | |

| MappedToFwPort | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | imaFlex CXP-12 Quad:[0:3]<br>imaFlex CXP-12 Penta:[0:4] |
| The firmware CXP channel which is currently monitored by the module. Note that there is not necessarily a one-by-one mapping between firmware port (resource CameraPort) and frame grabber port (physical connector). Instead it can be any permutation. The software discovery process reorders the channels and ports to achieve correct virtual interconnect. | |

| Decoder8b10bError | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 2^48-1] |
| Number of measured symbols received by the channel transceiver, which are not in 8b10b encoding or/and have wrong disparity. | |

| ByteAlignment8b10bLocked | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO, YES} |
| This parameter shows whether the channel transceiver found the byte boundaries or not, so decoding the of 8b10b input stream can be performed. | |

| CurrentPortBitRate | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 12.5] |
| This parameter shows the bit rate of the channel in Gb/s which is currently configured in the transceiver by the runtime discovery software. | |

| StreamPacketSize | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 8192 |
| **Range** | [4 : 65532] |

| **StreamPacketSize** |
|---|
| This parameter shows the supported maximum stream packet size, which will be written also to the CXP camera during the CXP discovery step. The value is set by the firmware and complies with the CXP standard. This means the value must be divisible by 4 bytes because all packets are 32 bit aligned. |

| **CxpStandard** | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | CXP_1_1_1 |
| **Range** | {CXP_1_0, CXP_1_1_1, CXP_2_0, unknown} |
| This parameter shows the version of the CXP standard which is currently in use. | |

| **RxTriggerOverRequestCount** | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts the received trigger packets for which no acknowledgment could be sent, because the acknowledgment of the previous trigger was still ongoing. 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. | |

| **TxTriggerOverRequestCount** | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts the trigger requests which 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. If multiple trigger requests occur at the same time on parallel ports of a single *CxpTxTrigger* operator, the parameter might not count each skipped trigger. This can also happen if the requests occur within only a few clock cycles (e.g. < 10), which leads to multiple skipped trigger requests within a few clock cycles. | |

| **RxTriggerAckLostCount** | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts the situations in which a trigger packet was sent, but no acknowledgment packet was received for it yet and the timeout (480ns for 1-6Gb/s, 240ns for 10-12.5Gb/s) is reached. 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. | |

| **RxGpioOverRequestCount** | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which a GPIO packet was received but no acknowledgment could be sent, because the transmitter still was busy sending the acknowledgment of the previously received GPIO packet. See CXP 1.0 standard, chapter 8.3.3. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| **TxGpioOverRequestCount** | |
|---|---|
| **Type** | Dynamic read parameter |

| TxGpioOverRequestCount | |
|---|---|
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts GPIO requests which were skipped, because the transmitter was still busy by sending the previous GPIO packet. See CXP 1.0 standard, chapter 8.3.3. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxGpioAckLostCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which a GPIO packet was sent but no acknowledgment packet was received for it yet and the timeout (480ns for 1-6Gb/s, 240ns for 10-12.5Gb/s) is reached. See CXP 1.0 standard, chapter 8.3.3. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxStreamIncompleteCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which a received stream packet is not correctly formatted, e.g. it misses the end of packet indicator etc. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxControlAckLostCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| 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. | |

| RxControlTagErrorCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which an acknowledgment for a control packet was received with a tag which is not matching the expected tag sent in the correspondent 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. | |

| RxControlAckIncompleteCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which an acknowledgment for a control packet was received which was not correctly formatted, e.g. it misses the end of packet indicator etc. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxEventOverRequestCount | |
|---|---|
| **Type** | Dynamic read parameter |

| RxEventOverRequestCount | |
|---|---|
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which another event packet is received but the corresponding acknowledgment can't be sent because the transmitter is still sending a previous event acknowledgment. See CXP standard 2.0, chapter 9.8.1.1. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxEventIncompleteCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which the received event 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. | |

| RxHeartBeatIncompleteCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| 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. | |

| RxUnknownDataReceivedCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which the unknown packet data is received, which is not part of the CXP standard. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxErrorCorrectedCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts errors received in packet headers and trailers which could be corrected. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxTriggerPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received trigger packets. | |

| RxTriggerAckPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received trigger acknowledge packets. | |

| RxGpioPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received GPIO packets. | |

| RxGpioAckPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received GPIO acknowledge packets. | |

| RxStreamPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received stream packets. | |

| RxControlAckPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received stream acknowledge packets. | |

| RxLinkTestPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received link test packets. | |

| RxEventPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received event packets. | |

| RxHeartBeatPacketCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether errors were corrected in received heart beat packets. | |

| RxUncorrectableErrorCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts errors received in packet headers and trailers which could not be corrected. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxTriggerPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received trigger packets which could not be corrected. | |

| RxTriggerAckPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received trigger acknowledgement packets which could not be corrected. | |

| RxGpioPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received GPIO packets which could not be corrected. | |

| RxGpioAckPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received GPIO acknowledgement packets which could not be corrected. | |

| RxStreamPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received stream packets which could not be corrected. | |

| RxControlAckPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received control acknowledgement packets which could not be corrected. | |

| RxLinkTestPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received link test packets which could not be corrected. | |

| RxEventPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received event packets which could not be corrected. | |

| RxHeartBeatPacketNotCorrected | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were errors in received heart beat packets which could not be corrected. | |

| RxPacketCrcErrorCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts CRC errors in received packets. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxStreamPacketCrcError | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| Notify whether there were CRC errors in received stream packets. | |

| RxControlAckPacketCrcError | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were CRC errors in received control acknowledgement packets. | |

| RxEventPacketCrcError | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter notifies whether there were CRC errors in received event packets. | |

| RxUnsupportedPacketCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts received unsupported packets, i.e. those which are not allowed by CXP standard. For example event/heart beat packets in CXP 1.x or GPIO packets in CXP 2.0 and CXP 1.1.x. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxUnsupportedGpioPacketReceived | |
|---|---|
| **Type** | Dynamic read parameter |

| RxUnsupportedGpioPacketReceived | |
|---|---|
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter indicates whether a GPIO packet was received while using a CXP standard higher than 1.0. | |

| RxUnsupportedEventPacketReceived | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter indicates whether an event packet was received while using a CXP standard less than 2.0. | |

| RxUnsupportedHeartBeatPacketReceived | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter indicates whether a heart beat packet was received while using a CXP standard less than 2.0. | |

| RxUnsupportedGpioAckPacketReceived | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter indicates whether a GPIO acknowledgment was received while using a CXP standard higher than 1.0. | |

| RxUnsupportedGpioPacketRequestReceived | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter indicates whether a GPIO request from VisualApplets was received while using a CXP standard higher than 1.0. | |

| RxLengthErrorCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts situations in which the length of the packet does not match the length defined in the packet header. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxStreamPacketLengthError | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter indicated whether there is a length error in the stream packets. | |

| RxEventPacketLengthError | |
|---|---|
| **Type** | Dynamic read parameter |

| RxEventPacketLengthError | |
|---|---|
| **Default** | NO |
| **Range** | {NO,YES} |
| This parameter indicated whether there is a length error in event packets. | |

| RxEventTagErrorCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| This parameter counts tag errors in received event packets, which happens when the event packet tag got a gap. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

| RxStreamPacketCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 2^31-1] |
| 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. | |

| RxHeartBeatMaxPeriodViolationCount | |
|---|---|
| **Type** | Dynamic read parameter |
| **Default** | 0 |
| **Range** | [0 : 8191] |
| The heart beat period is defined in CXP 2.0 standard as 100ms maximum, i.e. within that time at least 1 heart beat packet needs to be send by the camera. This parameter counts the situations in which this condition is not met. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. | |

# 29.33. Operator CxpRxTrigger

**Operator Library: Hardware Platform**

This operator provides received CXP trigger data to the VisualApplets design.

In CXP 2.0, the Rx-trigger is a 4-bit value with custom defined interpretation. Every received trigger packet causes the operator to output one 4-bit data word of a 0D data stream.

In CXP 1.x, the Rx-trigger is a 1-bit value. In this case, the operator also provides a 4-bit interface. However, bit[0] is used for encoding the reception of CXP 1.x trigger packets, and bits [1:3] are always zero. The rising edge of a trigger leads to the output of one data word with value 1 and the falling edge will lead to the the output of one data word with value 0.

The operator is using one resource *CxpRxTriggerPort*, which is in range {0, 1, 2, 3} for imaFlex CXP-12 Quad, and in range {0, 1, 2, 3, 4} for imaFlex CXP-12 Penta. This resource corresponds to the CXP channel in which the trigger packets are received.

| Available for Hardware Platform |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |

## 29.33.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, trigger data |

## 29.33.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 4 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALD_PIXEL0D |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.33.3. Parameters

None

## 29.33.4. Examples of Use

The use of operator CxpRxTrigger is shown in the following examples:

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 29.34. Operator CxpTxTrigger

**Operator Library: Hardware Platform**

This operator generates trigger packets according to signal edges detected at the input port.

There are two possible implementations of the *CxpTxTrigger* operator that depend on the number of input ports:

- Single Input Port: The Tx Trigger specification from CXP 1.X standard is used. *CXPLinkTrigger0* is sent with every rising edge on the input port *I*. When the *TxTriggerPacketMode* parameter is not set to *RisingEdgeOnly*, *CXPLinkTrigger1* is sent with every falling edge on the input port *I*. If the operator is instantiated with a single input port, the name of said port is *I*.
- Multiple Input Ports: The Tx Trigger specification from CXP 2.X standard is used (LSB of delay is used to implement trigger 2 and 3). Every input port is assigned to one of the *CXPLinkTriggers* from 0 to 3. Every rising edge on an input port will result in the corresponding *CXPLinkTrigger* packet being sent. For this configuration the *TxTriggerPacketMode* parameter is deactivated. If the operator is instantiated with multiple input ports, the names of the input ports range from *LinkTrigger0* to *LinkTrigger3* depending on the number of configured input ports.

The operator uses one resource *CxpTxTriggerPort*, which is in range {0, 1, 2, 3} for imaFlex CXP-12 Quad, and in range {0, 1, 2, 3, 4} for imaFlex CXP-12 Penta. This resource corresponds to the CXP channel in which the trigger packets are sent.

Since a rising edge (and potentially falling edge for single input ports) on the input ports leads to a full trigger packet being sent, there must be a gap between edges. If the gap between edges is not large enough it leads to 'over-triggering'. This means only the first edge results in a trigger packet, while the suceeding edges are ignored. The *CxpPortStatus* operator contains the parameter *TxTriggerOverRequestCount*, which counts the number of times that over-triggering has occurred. If multiple trigger edges are lost in a very short period of time due to over-triggering (e.g. trigger edges on parallel input ports), the *TxTriggerOverRequestCount* might not count the exact number of lost trigger edges. Instead, the multiple over-triggers will lead to a single increment of *TxTriggerOverRequestCount*.

| Available for Hardware Platform |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |

## 29.34.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, n == 1, trigger input<br>LinkTrigger[n], n > 1, trigger input |

## 29.34.2. Supported Link Format

| Link Parameter | Input Link I, n == 1 | Input Link LinkTrigger[n], n > 1 |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |

| Link Parameter | Input Link I, n == 1 | Input Link LinkTrigger[n], n > 1 |
|---|---|---|
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 29.34.3. Parameters

| TxTriggerPacketMode | |
|---|---|
| **Type** | Dynamic Write parameter |
| **Default** | RisingFallingEdge |
| **Range** | {RisingFallingEdge,RisingEdgeOnly} |
| This parameter defines for which trigger input edge a trigger packet is sent. If you select the *RisingFallingEdge* mode, a packet is sent for rising and falling edges. If you select the *RisingEdgeOnly* mode, only packets are sent for the rising edge. This parameter is only active, when only a single input port is used. | |

### 29.34.4. Examples of Use

The use of operator CxpTxTrigger is shown in the following examples:

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 29.35. Operator RS485

**Operator Library: Hardware Platform**

This operator provides an interface to the I/O signals of the corresponding RS-485 drivers for the GPIO connector of the framegrabber.




The operator is using one resource of type RS485Port. The resource index is always 0, because there is only one RS-485 interface.

## 29.35.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | DI, transmit data input<br>DE, data enable for transmission |
| **Output Link** | RO, received data output |

## 29.35.2. Supported Link Format

| Link Parameter | Input Link DI | Input Link DE | Output Link RO |
|---|---|---|---|
| **Bit Width** | 1 | 1 | 1 |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any | any |
| **Max. Img Height** | any | any | any |

## 29.35.3. Parameters

None

# 29.36. Operator CXPDualCamera

**Operator Library: Hardware Platform**

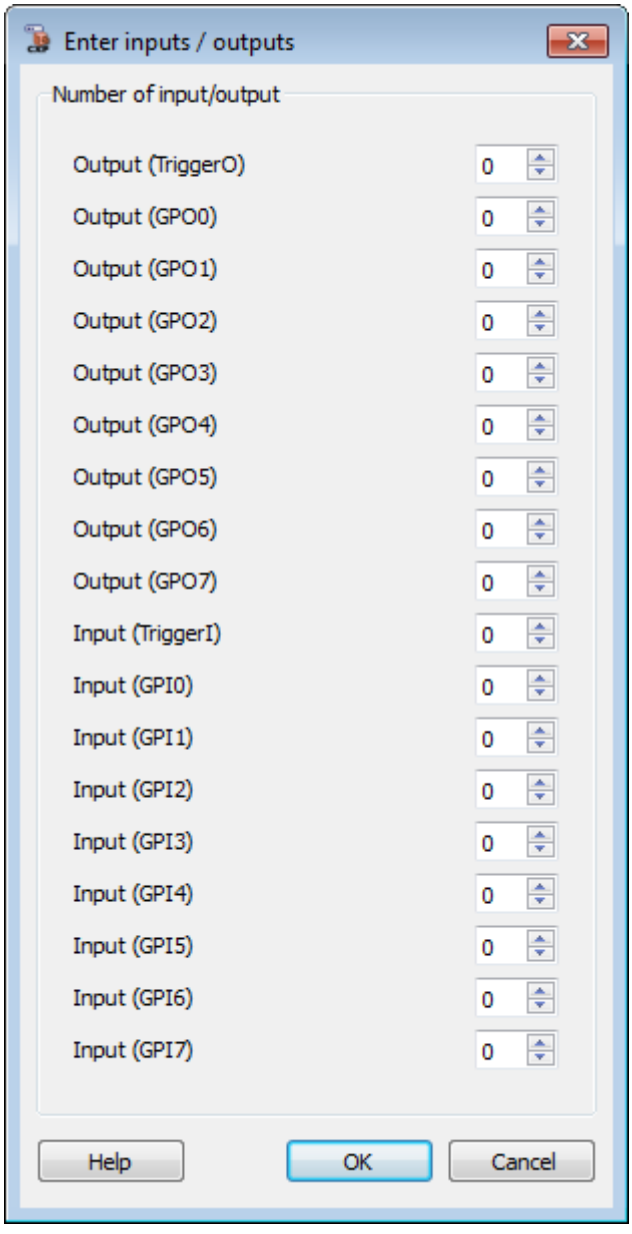
This operator represents the image data interface between a CXP dual channel camera and VisualApplets.

| Available for Hardware Platforms |
|---|
| mE5 marathon VCX-QP |
| mE5 ironman VQ8-CXP6D |
| mE5 ironman VQ8-CXP6B |

The operator provides image data on its output O. This output is always present.

In addition to this standard output port, you can specify various optional ports if required by your design.

The following pop-up dialog appears during operator instantiation:

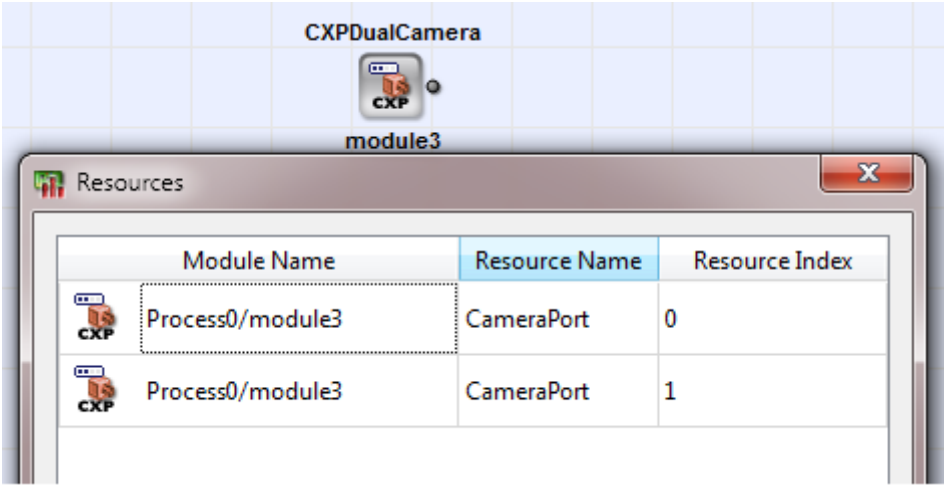


Here, you can specify the optional ports for general purpose inputs and outputs over CXP cable, as well as optional trigger ports to and from the camera.

You can define the availability of the particular GPIs (General Purpose Inputs), the particular GPOs (General Purpose Outputs), the trigger input port, and the trigger output port.

If you set the port availability to "0" (default), the port will not be present in the operator. If you set the port availability to 1, the particular port is available at the operator interface.

The operator uses 2 resources of type *CameraPort*.




**Modifying Image Width and Image Height**

You can modify width and height to the camera-specific settings. However, the maximal image width on operator port O **must be divisible by the parallelism** of port O.

**Make sure the maximal image width is divisible by the parallelism of port O!**

**Value for Parallelism**

The parallelism on port O needs to be set at least to the minimum value (stated in parameter *MinimalParallelism*). A lower value for the parallelism is not allowed. The minimal parallelism is calculated by VisualApplets on the basis of the values you define for other parameters of this operator. This way, incoming data can always be received.

**Bit Width and Format Type**

The bit width on port O (image port) depends on the selected format type (parameter *FormatType*):

- If FormatType = Gray: Bit Width can have any value (VAF_Gray and FL_NONE).
- If FormatType = RGB: Bit Width is a multiple of 3 (VAF_Color und FL_RGB)
- If FormatType = RGBA: Bit Width is a multiple of 4 (VAF_Gray and FL_NONE)
- If FormatType = RAW: Bit Width is 32 (VAF_Gray und FL_NONE)

## 29.36.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |

| Property | Value |
|---|---|
| **Input Links** | - TriggerI (optional), Trigger sent from frame grabber to camera over CXP channel. This port is often used to trigger line scan cameras.<br>- GPIx (optional), General purpose input [x] sent from the frame grabber to camera over CXP channel. Available are GPI0, GPI1, GPI2, GPI3, GPI4, GPI5, GPI6, and GPI7. |
| **Output Links** | - O, image data output<br>- TriggerO (optional), Trigger sent from the camera to the frame grabber over CXP channel.<br>- GPOx (optional), General purpose output [x] sent from the camera to the frame grabber over CXP channel. Available are GPO0, GPO1, GPO2, GPO3, GPO4, GPO5, GPO6, and GPO7. |

### 29.36.2. Supported Link Format

| Link Parameter | Output Link - O | Input Link - TriggerI (optional) | Output Link - TriggerO (optional) |
|---|---|---|---|
| **Bit Width** | any | 1 | 1 |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | any❶ | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} (default: VALT_IMAGE2D) | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | {VAF_GRAY, VAF_COLOR}<br><br>VAF_GRAY = FL_NONE<br><br>VAF_COLOR = FL_RGB<br><br>(If Color Format is VAF_GRAY, Color Flavor is FL_NONE; if Color Format is VAF_COLOR, Color Flavor is FL_RGB.) | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | {FL_NONE, FL_RGB}<br><br>VAF_GRAY = FL_NONE<br><br>VAF_COLOR = FL_RGB<br><br>(If Color Format is VAF_GRAY, Color Flavor is FL_NONE; if Color Format is VAF_COLOR, Color Flavor is FL_RGB.) | FL_NONE | FL_NONE |
| **Max. Img Width** | any (default: 1024) | 1 | 1 |
| **Max. Img Height** | any (default: 1024) | 1 | 1 |

| Link Parameter | Input Link - GPIx (optional) | Output Link - GPOx (optional) |
|---|---|---|
| **Bit Width** | 1 | 1 |

| Link Parameter | Input Link - GPIx (optional) | Output Link - GPOx (optional) |
|---|---|---|
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | 1 | 1 |
| **Max. Img Height** | 1 | 1 |

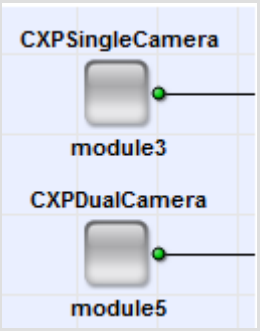
❶ Needs to be set at least to the minimum value stated in parameter *MinimalParallelism*.

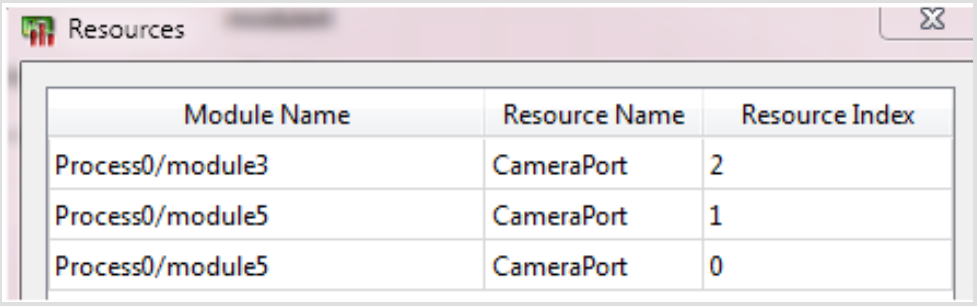
## 29.36.3. Parameters

| **CameraID** | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | {-1, 0, 1, 2, 3} |

The **CameraID** parameter defines the CXP master port ID to the firmware of the camera operator to allow automatic CXP topology discovery (camera-to-applet and applet-to-camera).

By using this parameter, you can specify the ID directly here in the operator. However, we recommend to specify the resources of the operator in the Recources dialog. (To open the Recources dialog, simply highlight the camera operator and select from the Design menu the menu item Resources.) The CameraID parameter will automatically update to the correct master port, using the resource dialog parameters. The following example is using one single camera and one dual camera:




The CameraID (CXP master port ID) of module3 (single camera) is 2. The CameraID (CXP master port ID) of module5 (dual camera) is 1.

| **CameraID** |
|---|
| The dual camera requires 2 CXP camera ports. (A quad camera requires 4 ports.) Only the master port is reflected in the CameraID parameter of the operator.<br><br>When you define more CameraPort resources than possible (the maximum is 4), e.g., when you instantiate 5 "CXPSingleCamera"" operators, the camera operator which requests the CameraPort resource last (while all four available ones are already occupied) sets its CameraID to -1. In this case, the design rule check reports an error.<br><br>The parameter CameraID is in range {-1,0,1,2,3}. |

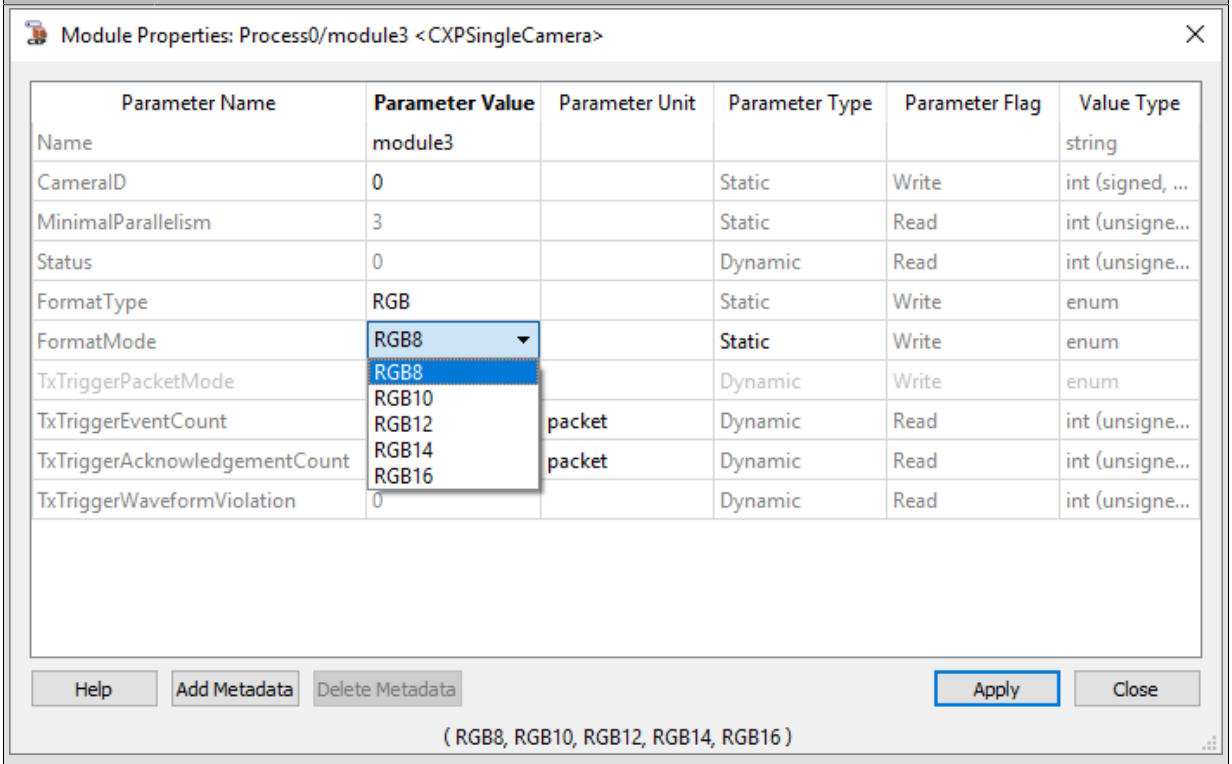
| **MinimalParallelism** | |
|---|---|
| **Type** | static read parameter |
| **Default** | 12 |
| **Range** | [1 : 512] |
| Minimal parallelism for the output link O that still allows to transport the maximal bandwidth of the camera without losing data. The minimal parallelism is calculated automatically.<br><br>The value depends on the pixel format you select in parameter *FormatMode*.<br><br>If you define parameter *FormatMode* to be a dynamic parameter (see description of parameter *FormatMode* below), the smallest minimal parallelism that might occur is calculated. | |

| **Status** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 2^7-1} |
| The **Status** parameter is a runtime, read-only parameter to reflect the current status of the camera operator. Bit[0] signalizes CXP stream packet loss detection. Bit[1] signalizes single byte error correction in CXP stream packets. Bit[2] signalizes multiple byte error detection in CXP stream packets. Bit[3..6] are reserved. This parameter might change in future versions. | |

| **FormatType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | GRAY |
| **Range** | GRAY, RGB, RGBA, RAW |
| Here, you can select which kind of pixel format (as defined by the CXP specification) you want to receive. The value you select here defines which pixel formats can be selected in parameter *FormatMode*. Hence, the setting of this parameter directly influences parameter *FormatMode*. See also documentation of parameter *FormatMode* below. | |

| **FormatMode** | |
|---|---|
| **Type** | static or dynamic (user-defined) write parameter |
| **Default** | Mono8 |
| **Range** | {Mono8, Mono10, Mono12, Mono14, Mono16, BayerGR8, BayerGR10, BayerGR12, BayerGR14, BayerGR16, BayerRG8, BayerRG10, BayerRG12, BayerRG14, BayerRG16, BayerGB8, BayerGB10, BayerGB12, BayerGB14, BayerGB16, BayerBG8, BayerBG10, BayerBG12, BayerBG14, BayerBG16, RGB8, RGB10, RGB12, RGB14, RGB16, RGBA8, RGBA10, RGBA12, RGBA14, RGBA16, Raw} |
| The parameter offers all pixel formats (as defined by the CXP specification) that belong to the format type you selected in in parameter *FormatType*. | |

**FormatMode**




The pixel formats you can select here depend on the setting of parameter *FormatType*. Hence, you always have only a pre-set of the full value range available.

To get another set of pixel formats, change the setting of parameter *FormatType*. See also documentation of parameter *FormatType* above.

In detail, the following settings of parameter *FormatType* provide the following pixel formats in parameter *FormatMode*:

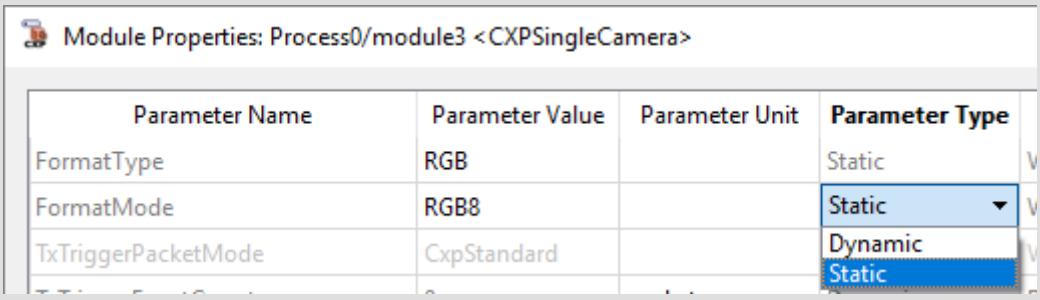
**FormatType == GRAY:** {Mono8, Mono10, Mono12, Mono14, Mono16, BayerGR8, BayerGR10, BayerGR12, BayerGR14, BayerGR16, BayerRG8, BayerRG10, BayerRG12, BayerRG14, BayerRG16, BayerGB8, BayerGB10, BayerGB12, BayerGB14, BayerGB16, BayerBG8, BayerBG10, BayerBG12, BayerBG14, BayerBG16}

**FormatType == RGB:** {RGB8, RGB10, RGB12, RGB14, RGB16}

**FormatType == RGBA:** {RGBA8, RGBA10, RGBA12, RGBA14, RGBA16}

**FormatType == RAW:** {Raw} Pixel format Raw has a fix bit width of 32Bit. The data are interpreted as received.

You can decide if you want to use this parameter as a dynamic or as a static parameter. Select in column Parameter Type:



| TxTriggerPacketMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | CxpStandard |
| **Range** | {CxpStandard, RisingEdgeOnly} |

With parameter *TxTriggerPacketMode* you can increase the bandwidth of the CXP cables that connect the frame grabber with the camera. The increase of bandwidth is achieved by a slight modification of the camera trigger signal interpretation.

Please note that this feature can only be used with specific cameras that support this feature. For details, please contact your local distributor or the Basler Support [https://www.baslerweb.com/en/sales-support/support-contact/].

Parameter *TxTriggerPacketMode* allows two values (CxpStandard, RisingEdgeOnly):

- **CxpStandard**: The recommended setting is value CxpStandard. If set to value CxpStandard, the camera trigger works in accordance with the CXP specification. (The CXP specification defines that the frame grabber sents two packets via CXP cable to the camera in order to trigger the camera once.)
- **RisingEdgeOnly**: If set to value RisingEdgeOnly, the frame grabber sents only one packet to the camera in order to trigger the camera once. The result is an increased bandwidth and a higher line rate.

| TxTriggerEventCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0;2^20-1} |

The **TxTriggerEventCount** parameter indicates how many trigger edge events have been sent to the camera.

| TxTriggerAcknowledgementCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0;2^20-1} |

The **TxTriggerAcknowledgementCount** parameter indicates how many trigger acknowledgement packets sent by the camera (in answer to the trigger edge packets sent before) have been received by the frame grabber.

| TxTriggerWaveformViolation | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0,1} |

The parameter is set to 1 by the camera operator if the operator detects a distance between two trigger edges on port TriggerI which violates the minimal edge frequency. The minimal edge frequency is 550 ns (nanoseconds) on all microEnable 5 platforms. The parameter holds its value until it has been read. After being read, the parameter updates the value. Frequency control is running permanently and is not influenced by the read status of the parameter.

## 29.36.4. Examples of Use

The use of operator CXPDualCamera is shown in the following examples:

- Section 10.4.1, 'CoaXPress Area Scan Cameras'

  Tutorial - Basic Acquisition

- Section 10.4.1.2, 'Basic Acquisition Examples for two Dual Line CoaXPress Area Scan Cameras'
  Tutorial - Basic Acquisition
- Section 10.4.2, 'CoaXPress Line Scan Cameras'
  Tutorial - Basic Acquisition
- Section 10.4.2.2, 'Basic Acquisition Examples for two Dual Line CoaXPress Line Scan Cameras'
  Tutorial - Basic Acquisition

# 29.37. Operator CXPQuadCamera

**Operator Library: Hardware Platform**

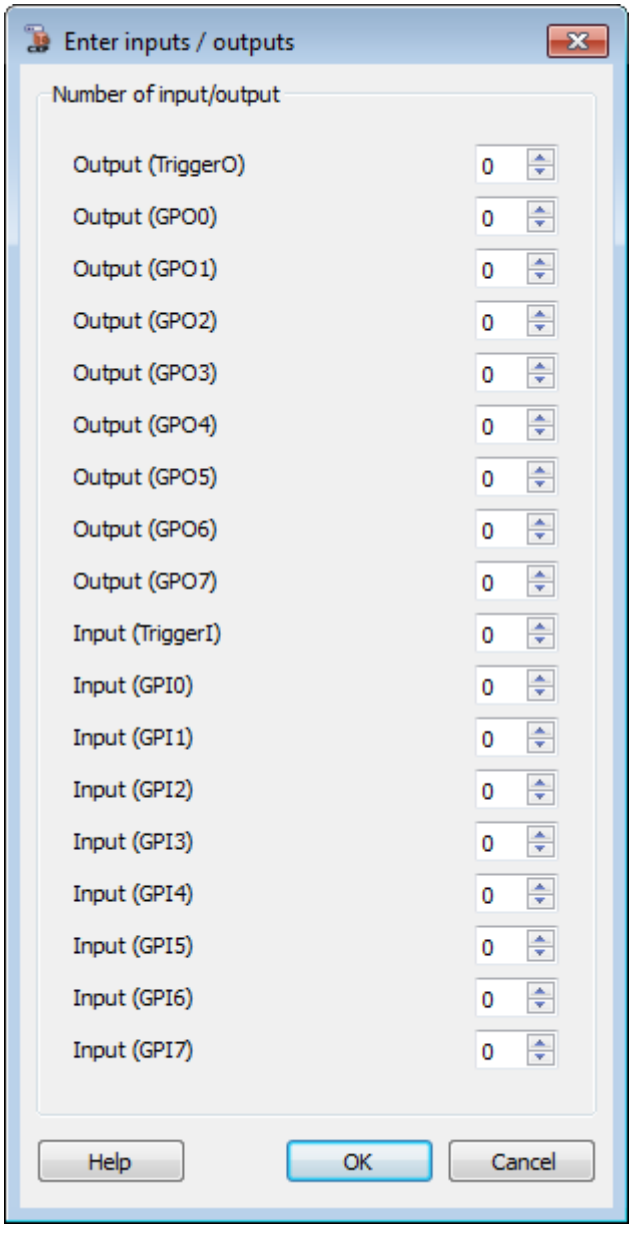
This operator represents the image data interface between a CXP quad channel camera and VisualApplets.

| Available for Hardware Platforms |
|---|
| mE5 marathon VCX-QP |
| mE5 ironman VQ8-CXP6D |
| mE5 ironman VQ8-CXP6B |

The operator provides image data on its output O. This output is always present.

In addition to this standard output port, you can configure a set of optional inputs and outputs.

The following pop-up dialog appears during operator instantiation:

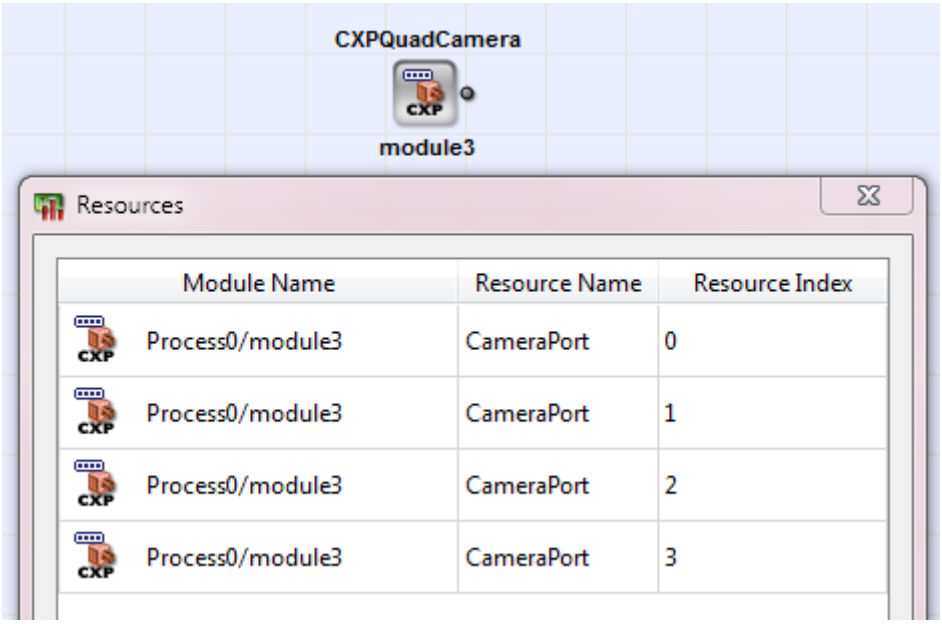


Here, you can specify the optional ports for general purpose inputs and outputs over CXP cable, as well as optional trigger ports to and from the camera.

You can define the availability of the particular GPIs (General Purpose Inputs), the particular GPOs (General Purpose Outputs), the trigger input port, and the trigger output port.

If you set the port availability to "0" (default), the port will not be present in the operator. If you set the port availability to 1, the particular port is available at the operator interface.

The operator uses 4 resources of type *CameraPort*.




**Modifying Image Width and Image Height**

You can modify width and height to the camera-specific settings. However, the maximal image width on operator port O **must be divisible by the parallelism** of port O.

**Make sure the maximal image width is divisible by the parallelism of port O!**

**Value for Parallelism**

The parallelism on port O needs to be set at least to the minimum value (stated in parameter *MinimalParallelism*). A lower value for the parallelism is not allowed. The minimal parallelism is calculated by VisualApplets on the basis of the values you define for other parameters of this operator. This way, incoming data can always be received.

**Bit Width and Format Type**

The bit width on port O (image port) depends on the selected format type (parameter *FormatType*):

- If FormatType = Gray: Bit Width can have any value (VAF_Gray and FL_NONE).
- If FormatType = RGB: Bit Width is a multiple of 3 (VAF_Color und FL_RGB)
- If FormatType = RGBA: Bit Width is a multiple of 4 (VAF_Gray and FL_NONE)
- If FormatType = RAW: Bit Width is 32 (VAF_Gray und FL_NONE)

### 29.37.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | - TriggerI (optional), Trigger sent from frame grabber to camera over CXP channel. This port is often used to trigger line scan cameras.<br>- GPIx (optional), General purpose input [x] sent from the frame grabber to camera over CXP channel. Available are GPI0, GPI1, GPI2, GPI3, GPI4, GPI5, GPI6, and GPI7. |
| **Output Links** | - O, image data output<br>- TriggerO (optional), Trigger sent from the camera to the frame grabber over CXP channel.<br>- GPOx (optional), General purpose output [x] sent from the camera to the frame grabber over CXP channel. Available are GPO0, GPO1, GPO2, GPO3, GPO4, GPO5, GPO6, and GPO7. |

### 29.37.2. Supported Link Format

| Link Parameter | Output Link - O | Input Link - TriggerI (optional) | Output Link - TriggerO (optional) |
|---|---|---|---|
| **Bit Width** | any | 1 | 1 |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | any❶ | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} (default: VALT_IMAGE2D) | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | {VAF_GRAY, VAF_COLOR}<br><br>VAF_GRAY = FL_NONE<br><br>VAF_COLOR = FL_RGB<br><br>(If Color Format is VAF_GRAY, Color Flavor is FL_NONE; if Color Format is VAF_COLOR, Color Flavor is FL_RGB.) | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | {FL_NONE, FL_RGB}<br><br>VAF_GRAY = FL_NONE<br><br>VAF_COLOR = FL_RGB<br><br>(If Color Format is VAF_GRAY, Color Flavor is FL_NONE; if Color Format is VAF_COLOR, Color Flavor is FL_RGB.) | FL_NONE | FL_NONE |
| **Max. Img Width** | any (default: 1024) | 1 | 1 |
| **Max. Img Height** | any (default: 1024) | 1 | 1 |

| Link Parameter | Input Link - GPIx (optional) | Output Link - GPOx (optional) |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | 1 | 1 |
| **Max. Img Height** | 1 | 1 |

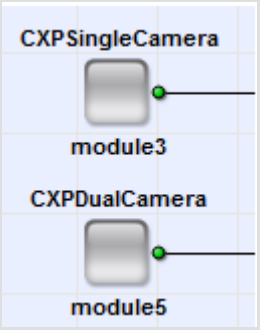
❶ Needs to be set at least to the minimum value stated in parameter *MinimalParallelism*.

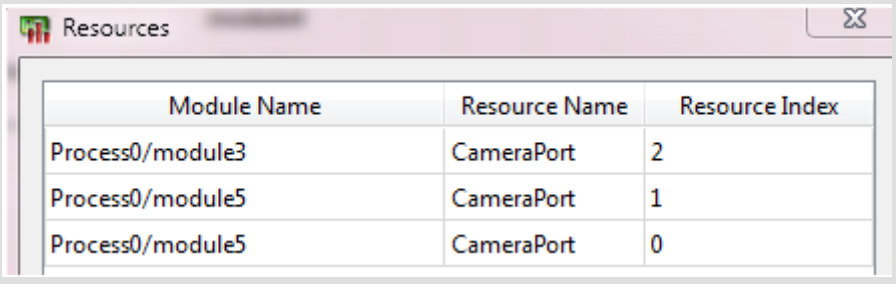
## 29.37.3. Parameters

| **CameraID** | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | {-1, 0, 1, 2, 3} |

The **CameraID** parameter defines the CXP master port ID to the firmware of the camera operator to allow automatic CXP topology discovery (camera-to-applet and applet-to-camera).

By using this parameter, you can specify the ID directly here in the operator. However, we recommend to specify the resources of the operator in the Recources dialog. (To open the Recources dialog, simply highlight the camera operator and select from the Design menu the menu item Resources.) The CameraID parameter will automatically update to the correct master port, using the resource dialog parameters. The following example is using one single camera and one dual camera:



| **CameraID** |
|---|
| The CameraID (CXP master port ID) of module3 (single camera) is 2. The CameraID (CXP master port ID) of module5 (dual camera) is 1.<br><br>The dual camera requires 2 CXP camera ports. (A quad camera requires 4 ports.) Only the master port is reflected in the CameraID parameter of the operator.<br><br>When you define more CameraPort resources than possible (the maximum is 4), e.g., when you instantiate 5 "CXPSingleCamera"" operators, the camera operator which requests the CameraPort resource last (while all four available ones are already occupied) sets its CameraID to -1. In this case, the design rule check reports an error.<br><br>The parameter CameraID is in range {-1,0,1,2,3}. |

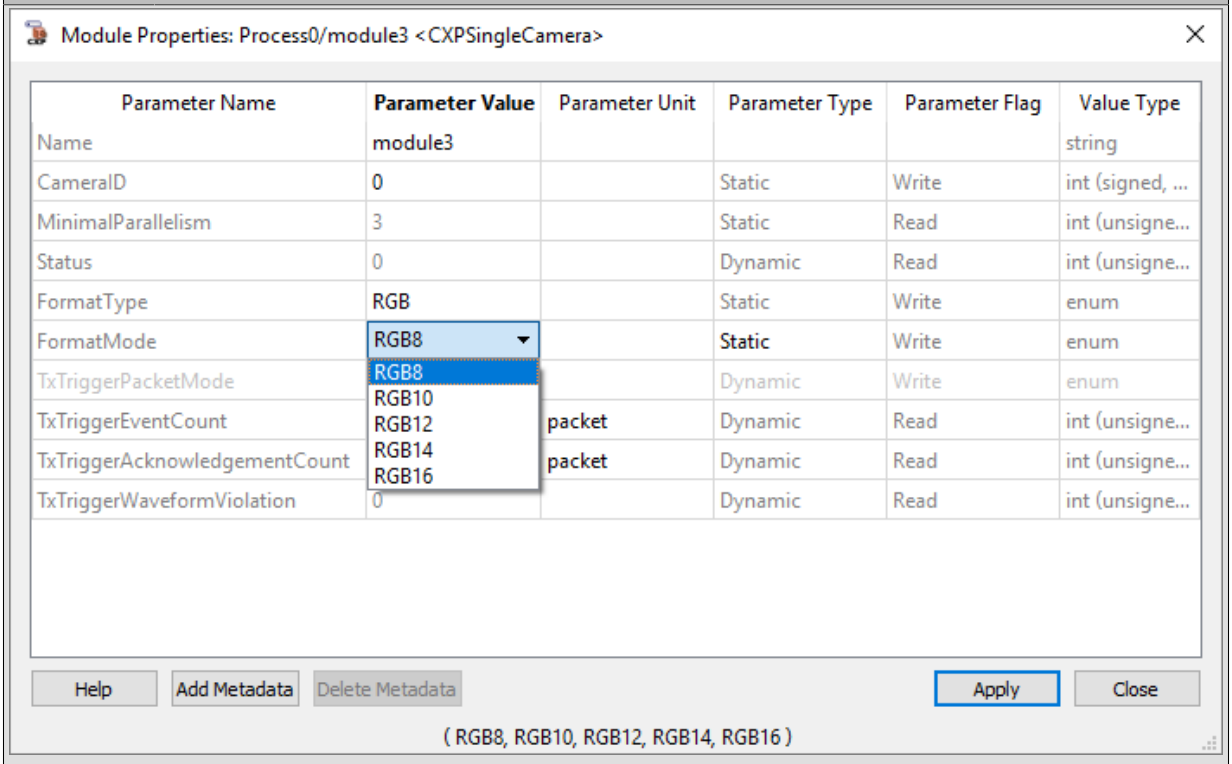
| **MinimalParallelism** | |
|---|---|
| **Type** | static read parameter |
| **Default** | 20 |
| **Range** | [1 : 512] |
| Minimal parallelism for the output link O that still allows to transport the maximal bandwidth of the camera without losing data. The minimal parallelism is calculated automatically.<br><br>The value depends on the pixel format you select in parameter *FormatMode*.<br><br>If you define parameter *FormatMode* to be a dynamic parameter (see description of parameter *FormatMode* below), the smallest minimal parallelism that might occur is calculated. | |

| **Status** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 2^7-1} |
| The **Status** parameter is a runtime, read-only parameter to reflect the current status of the camera operator. Bit[0] signalizes CXP stream packet loss detection. Bit[1] signalizes single byte error correction in CXP stream packets. Bit[2] signalizes multiple byte error detection in CXP stream packets. Bit[3..6] are reserved. This parameter might change in future versions. | |

| **FormatType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | GRAY |
| **Range** | GRAY, RGB, RGBA, RAW |
| Here, you can select which kind of pixel format (as defined by the CXP specification) you want to receive. The value you select here defines which pixel formats can be selected in parameter *FormatMode*. Hence, the setting of this parameter directly influences parameter *FormatMode*. See also documentation of parameter *FormatMode* below. | |

| **FormatMode** | |
|---|---|
| **Type** | static or dynamic (user-defined) write parameter |
| **Default** | Mono8 |
| **Range** | {Mono8, Mono10, Mono12, Mono14, Mono16, BayerGR8, BayerGR10, BayerGR12, BayerGR14, BayerGR16, BayerRG8, BayerRG10, BayerRG12, BayerRG14, BayerRG16, BayerGB8, BayerGB10, BayerGB12, BayerGB14, BayerGB16, BayerBG8, BayerBG10, BayerBG12, BayerBG14, BayerBG16, RGB8, RGB10, RGB12, RGB14, RGB16, RGBA8, RGBA10, RGBA12, RGBA14, RGBA16, Raw} |
| The parameter offers all pixel formats (as defined by the CXP specification) that belong to the format type you selected in in parameter *FormatType*. | |

**FormatMode**




The pixel formats you can select here depend on the setting of parameter *FormatType*. Hence, you always have only a pre-set of the full value range available.

To get another set of pixel formats, change the setting of parameter *FormatType*. See also documentation of parameter *FormatType* above.

In detail, the following settings of parameter *FormatType* provide the following pixel formats in parameter *FormatMode*:

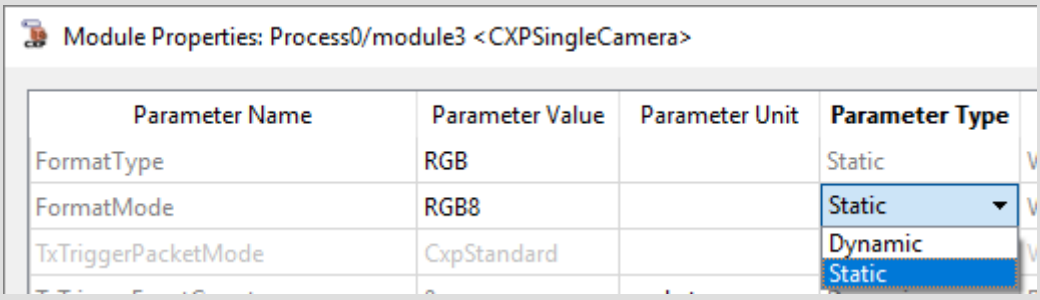
**FormatType == GRAY:** {Mono8, Mono10, Mono12, Mono14, Mono16, BayerGR8, BayerGR10, BayerGR12, BayerGR14, BayerGR16, BayerRG8, BayerRG10, BayerRG12, BayerRG14, BayerRG16, BayerGB8, BayerGB10, BayerGB12, BayerGB14, BayerGB16, BayerBG8, BayerBG10, BayerBG12, BayerBG14, BayerBG16}

**FormatType == RGB:** {RGB8, RGB10, RGB12, RGB14, RGB16}

**FormatType == RGBA:** {RGBA8, RGBA10, RGBA12, RGBA14, RGBA16}

**FormatType == RAW:** {Raw} Pixel format Raw has a fix bit width of 32Bit. The data are interpreted as received.

You can decide if you want to use this parameter as a dynamic or as a static parameter. Select in column Parameter Type:



| TxTriggerPacketMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | CxpStandard |
| **Range** | {CxpStandard, RisingEdgeOnly} |

With parameter *TxTriggerPacketMode* you can increase the bandwidth of the CXP cables that connect the frame grabber with the camera. The increase of bandwidth is achieved by a slight modification of the camera trigger signal interpretation.

This feature can only be used with specific cameras that support this feature. For details, contact your local distributor or the Basler Support department.

Parameter *TxTriggerPacketMode* allows two values (CxpStandard, RisingEdgeOnly):

- **CxpStandard**: The recommended setting is value CxpStandard. If set to value CxpStandard, the camera trigger works in accordance with the CXP specification. (The CXP specification defines that the frame grabber sents two packets via CXP cable to the camera in order to trigger the camera once.)
- **RisingEdgeOnly**: If set to value RisingEdgeOnly, the frame grabber sents only one packet to the camera in order to trigger the camera once. The result is an increased bandwidth and a higher line rate.

| TxTriggerEventCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0;2^20-1} |

The **TxTriggerEventCount** parameter indicates how many trigger edge events have been sent to the camera.

| TxTriggerAcknowledgementCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0;2^20-1} |

The **TxTriggerAcknowledgementCount** parameter indicates how many trigger acknowledgement packets sent by the camera (in answer to the trigger edge packets sent before) have been received by the frame grabber.

| TxTriggerWaveformViolation | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0,1} |

The parameter is set to 1 by the camera operator if the operator detects a distance between two trigger edges on port TriggerI which violates the minimal edge frequency. The minimal edge frequency is 550 ns (nanoseconds) on all microEnable 5 platforms. The parameter holds its value until it has been read. After being read, the parameter updates the value. Frequency control is running permanently and is not influenced by the read status of the parameter.

## 29.37.4. Examples of Use

The use of operator CXPQuadCamera is shown in the following examples:

- Section 10.4.1, 'CoaXPress Area Scan Cameras'

  Tutorial - Basic Acquisition
- Section 10.4.1.3, 'Basic Acquisition Examples for One Quad Line CoaXPress Area Scan Camera'

Tutorial - Basic Acquisition

- Section 10.4.2, 'CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition

- Section 10.4.2.3, 'Basic Acquisition Examples for One Quad Line CoaXPress Line Scan Camera'

  Tutorial - Basic Acquisition

# 29.38. Operator CXPSingleCamera

**Operator Library: Hardware Platform**

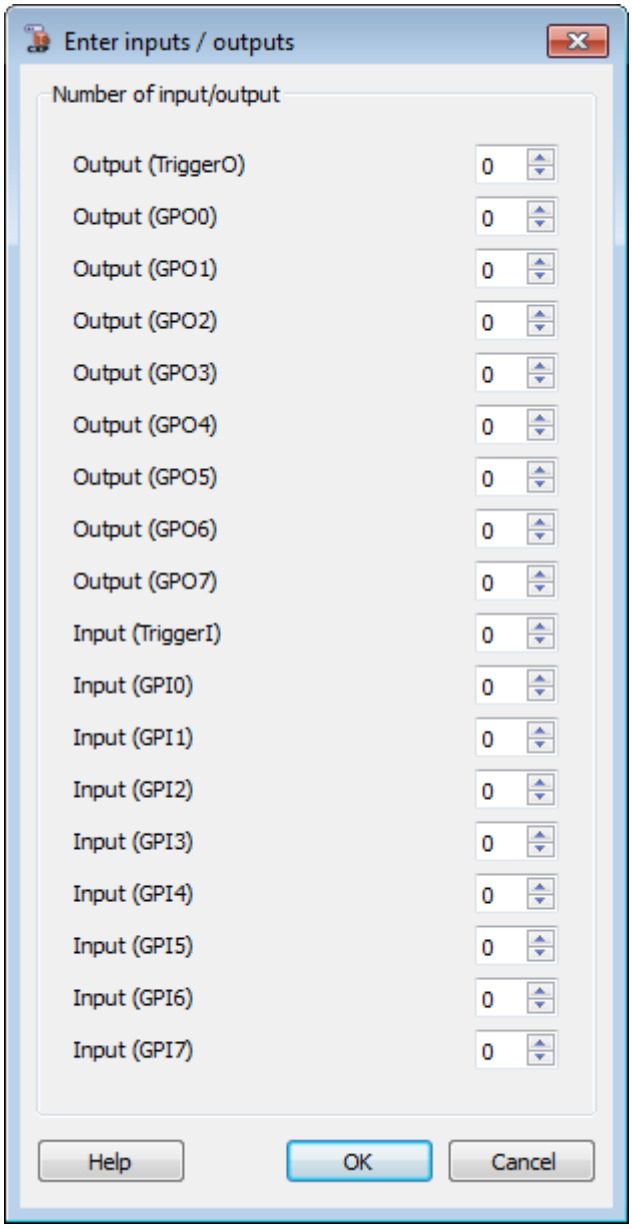
This operator represents the image data interface between a CXP single channel camera and VisualApplets.

| Available for Hardware Platforms |
|---|
| mE5 marathon VCX-QP |
| mE5 ironman VQ8-CXP6D |
| mE5 ironman VQ8-CXP6B |

The operator provides image data on its output O. This output is always present.

In addition to this standard output port, you can specify a set of optional inputs and outputs.

The following pop-up dialog appears during operator instantiation:

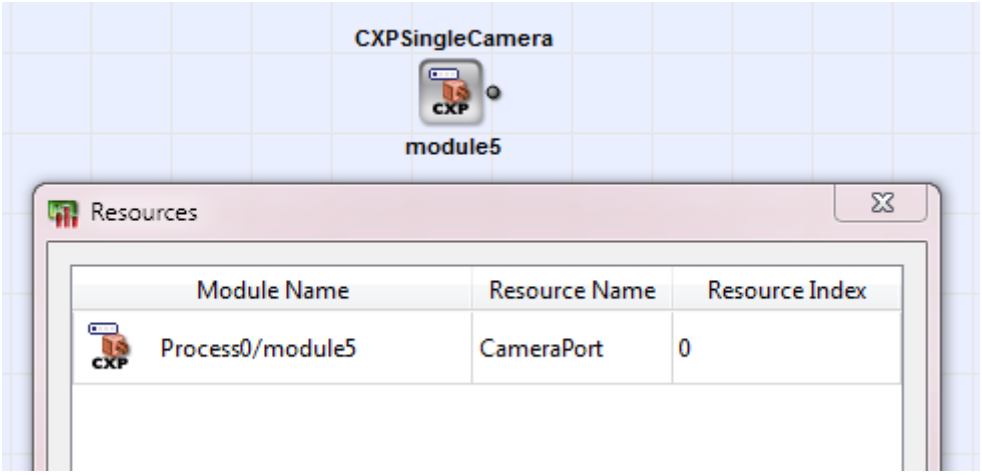



Here, you can specify the optional ports for general purpose inputs and outputs over CXP cable, as well as optional trigger ports to and from the camera.

You can define the availability of the particular GPIs (General Purpose Inputs), the particular GPOs (General Purpose Outputs), the trigger input port, and the trigger output port.

If you set the port availability to "0" (default), the port will not be present in the operator. If you set the port availability to 1, the particular port is available at the operator interface.

The operator uses 1 resource of type *CameraPort*.




**Modifying Image Width and Image Height**

You can modify width and height to the camera-specific settings. However, the maximal image width on operator port O **must be divisible by the parallelism** of port O.

**Make sure the maximal image width is divisible by the parallelism of port O!**

**Value for Parallelism**

The parallelism on port O needs to be set at least to the minimum value (stated in parameter *MinimalParallelism*). A lower value for the parallelism is not allowed. The minimal parallelism is calculated by VisualApplets on the basis of the values you define for other parameters of this operator. This way, incoming data can always be received.

**Bit Width and Format Type**

The bit width on port O (image port) depends on the selected format type (parameter *FormatType*):

- If FormatType = Gray: Bit Width can have any value (VAF_Gray and FL_NONE).
- If FormatType = RGB: Bit Width is a multiple of 3 (VAF_Color und FL_RGB)
- If FormatType = RGBA: Bit Width is a multiple of 4 (VAF_Gray and FL_NONE)
- If FormatType = RAW: Bit Width is 32 (VAF_Gray und FL_NONE)

## 29.38.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | - TriggerI (optional), Trigger sent from frame grabber to camera over CXP channel. This port is often used to trigger line scan cameras. |

| Property | Value |
|---|---|
| | - GPIx (optional), General purpose input [x] sent from the frame grabber to camera over CXP channel. Available are GPI0, GPI1, GPI2, GPI3, GPI4, GPI5, GPI6, and GPI7. |
| **Output Links** | - O, image data output<br>- TriggerO (optional), Trigger sent from the camera to the frame grabber over CXP channel.<br>- GPOx (optional), General purpose output [x] sent from the camera to the frame grabber over CXP channel. Available are GPO0, GPO1, GPO2, GPO3, GPO4, GPO5, GPO6, and GPO7. |

### 29.38.2. Supported Link Format

| Link Parameter | Output Link - O | Input Link - TriggerI (optional) | Output Link - TriggerO (optional) |
|---|---|---|---|
| **Bit Width** | any | 1 | 1 |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | any❶ | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} (default: VALT_IMAGE2D) | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | {VAF_GRAY, VAF_COLOR}<br><br>VAF_GRAY = FL_NONE<br><br>VAF_COLOR = FL_RGB<br><br>(If Color Format is VAF_GRAY, Color Flavor is FL_NONE; if Color Format is VAF_COLOR, Color Flavor is FL_RGB.) | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | {FL_NONE, FL_RGB}<br><br>VAF_GRAY = FL_NONE<br><br>VAF_COLOR = FL_RGB<br><br>(If Color Format is VAF_GRAY, Color Flavor is FL_NONE; if Color Format is VAF_COLOR, Color Flavor is FL_RGB.) | FL_NONE | FL_NONE |
| **Max. Img Width** | any (default: 1024) | 1 | 1 |
| **Max. Img Height** | any (default: 1024) | 1 | 1 |

| Link Parameter | Input Link - GPIx (optional) | Output Link - GPOx (optional) |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |

| Link Parameter | Input Link - GPIx (optional) | Output Link - GPOx (optional) |
|---|---|---|
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | 1 | 1 |
| **Max. Img Height** | 1 | 1 |

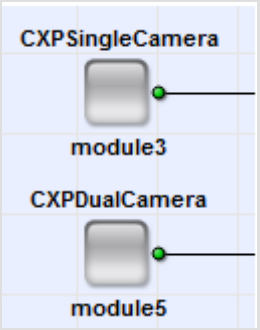
❶ Needs to be set at least to the minimum value stated in parameter *MinimalParallelism*.

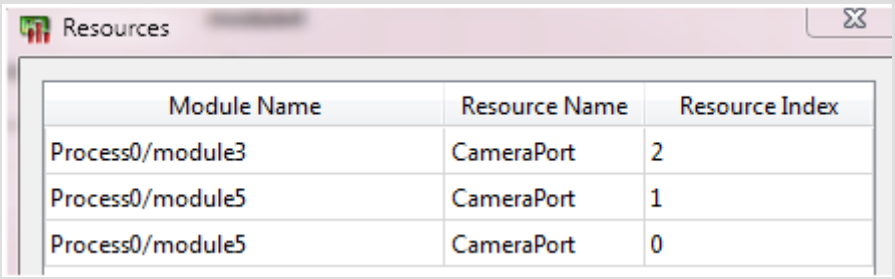
## 29.38.3. Parameters

| **CameraID** | |
|---|---|
| **Type** | static write parameter |
| **Default** | |
| **Range** | {-1, 0, 1, 2, 3} |

The **CameraID** parameter defines the CXP master port ID to the firmware of the camera operator to allow automatic CXP topology discovery (camera-to-applet and applet-to-camera).

By using this parameter, you can specify the ID directly here in the operator. However, we recommend to specify the resources of the operator in the Recources dialog. (To open the Recources dialog, simply highlight the camera operator and select from the Design menu the menu item Resources.) The CameraID parameter will automatically update to the correct master port, using the resource dialog parameters. The following example is using one single camera and one dual camera:




The CameraID (CXP master port ID) of module3 (single camera) is 2. The CameraID (CXP master port ID) of module5 (dual camera) is 1.

The dual camera requires 2 CXP camera ports. (A quad camera requires 4 ports.) Only the master port is reflected in the CameraID parameter of the operator.

| **CameraID** |
|---|
| When you define more CameraPort resources than possible (the maximum is 4), e.g., when you instantiate 5 "CXPSingleCamera"" operators, the camera operator which requests the CameraPort resource last (while all four available ones are already occupied) sets its CameraID to -1. In this case, the design rule check reports an error.<br><br>The parameter CameraID is in range {-1,0,1,2,3}. |

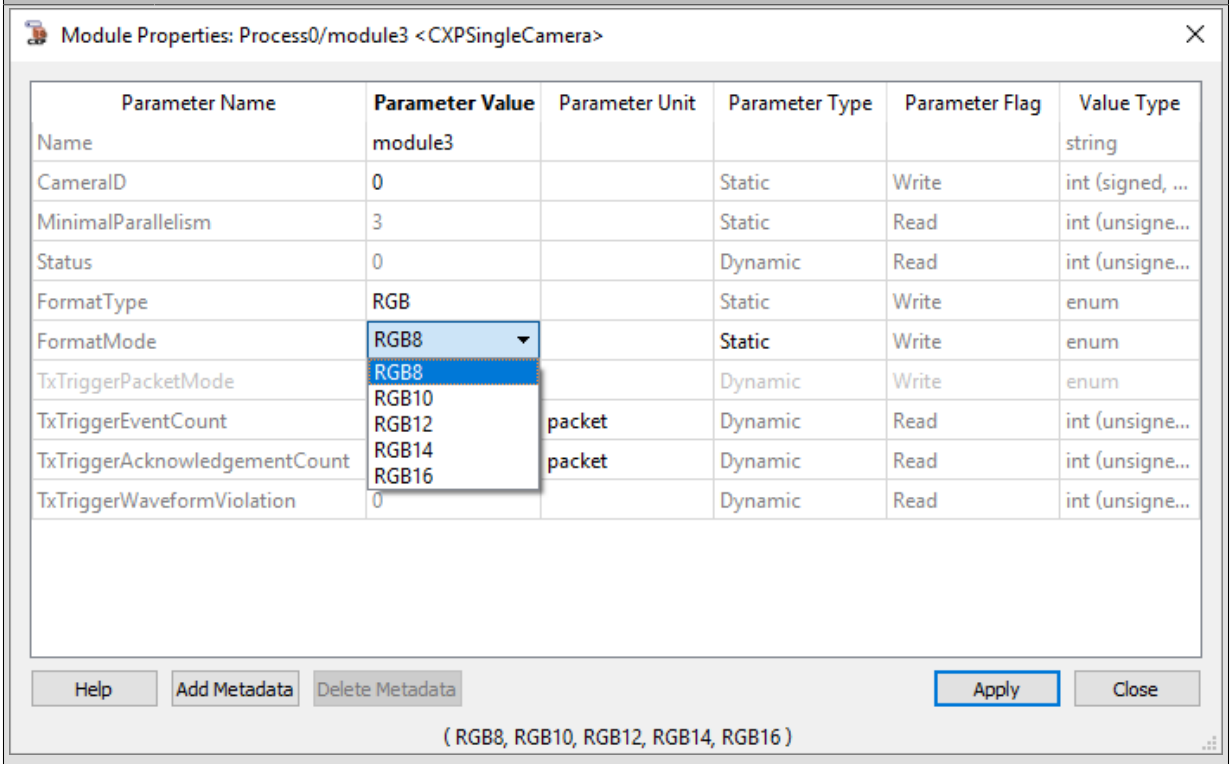
| **MinimalParallelism** | |
|---|---|
| **Type** | static read parameter |
| **Default** | 8 |
| **Range** | [1 : 512] |
| Minimal parallelism for the output link O that still allows to transport the maximal bandwidth of the camera without losing data. The minimal parallelism is calculated automatically.<br><br>The value depends on the pixel format you select in parameter *FormatMode*.<br><br>If you define parameter *FormatMode* to be a dynamic parameter (see description of parameter *FormatMode* below), the smallest minimal parallelism that might occur is calculated. | |

| **Status** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | {0; 2^7-1} |
| The **Status** parameter is a runtime, read-only parameter to reflect the current status of the camera operator. Bit[0] signalizes CXP stream packet loss detection. Bit[1] signalizes single byte error correction in CXP stream packets. Bit[2] signalizes multiple byte error detection in CXP stream packets. Bit[3..6] are reserved. This parameter might change in future versions. | |

| **FormatType** | |
|---|---|
| **Type** | static write parameter |
| **Default** | GRAY |
| **Range** | GRAY, RGB, RGBA, RAW |
| Here, you can select which kind of pixel format (as defined by the CXP specification) you want to receive. The value you select here defines which pixel formats can be selected in parameter *FormatMode*. Hence, the setting of this parameter directly influences parameter *FormatMode*. See also documentation of parameter *FormatMode* below. | |

| **FormatMode** | |
|---|---|
| **Type** | static or dynamic (user-defined) write parameter |
| **Default** | Mono8 |
| **Range** | {Mono8, Mono10, Mono12, Mono14, Mono16, BayerGR8, BayerGR10, BayerGR12, BayerGR14, BayerGR16, BayerRG8, BayerRG10, BayerRG12, BayerRG14, BayerRG16, BayerGB8, BayerGB10, BayerGB12, BayerGB14, BayerGB16, BayerBG8, BayerBG10, BayerBG12, BayerBG14, BayerBG16, RGB8, RGB10, RGB12, RGB14, RGB16, RGBA8, RGBA10, RGBA12, RGBA14, RGBA16, Raw} |
| The parameter offers all pixel formats (as defined by the CXP specification) that belong to the format type you selected in in parameter *FormatType*. | |

**FormatMode**




The pixel formats you can select here depend on the setting of parameter *FormatType*. Hence, you always have only a pre-set of the full value range available.

To get another set of pixel formats, change the setting of parameter *FormatType*. See also documentation of parameter *FormatType* above.

In detail, the following settings of parameter *FormatType* provide the following pixel formats in parameter *FormatMode*:

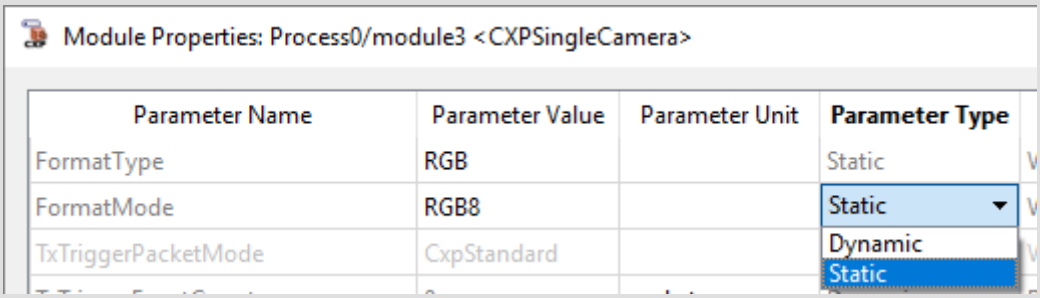
**FormatType == GRAY:** {Mono8, Mono10, Mono12, Mono14, Mono16, BayerGR8, BayerGR10, BayerGR12, BayerGR14, BayerGR16, BayerRG8, BayerRG10, BayerRG12, BayerRG14, BayerRG16, BayerGB8, BayerGB10, BayerGB12, BayerGB14, BayerGB16, BayerBG8, BayerBG10, BayerBG12, BayerBG14, BayerBG16}

**FormatType == RGB:** {RGB8, RGB10, RGB12, RGB14, RGB16}

**FormatType == RGBA:** {RGBA8, RGBA10, RGBA12, RGBA14, RGBA16}

**FormatType == RAW:** {Raw} Pixel format Raw has a fix bit width of 32Bit. The data are interpreted as received .

You can decide if you want to use this parameter as a dynamic or as a static parameter. Select in column Parameter Type:



| TxTriggerPacketMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | CxpStandard |
| **Range** | {CxpStandard, RisingEdgeOnly} |

With parameter *TxTriggerPacketMode* you can increase the bandwidth of the CXP cables that connect the frame grabber with the camera. The increase of bandwidth is achieved by a slight modification of the camera trigger signal interpretation.

Please note that this feature can only be used with specific cameras that support this feature. For details, please contact your local distributor or the Basler Support department.

Parameter *TxTriggerPacketMode* allows two values (CxpStandard, RisingEdgeOnly):

- **CxpStandard**: The recommended setting is value CxpStandard. If set to value CxpStandard, the camera trigger works in accordance with the CXP specification. (The CXP specification defines that the frame grabber sents two packets via CXP cable to the camera in order to trigger the camera once.)
- **RisingEdgeOnly**: If set to value RisingEdgeOnly, the frame grabber sents only one packet to the camera in order to trigger the camera once. The result is an increased bandwidth and a higher line rate.

| TxTriggerEventCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0;2^20-1} |

The **TxTriggerEventCount** parameter indicates how many trigger edge events have been sent to the camera.

| TxTriggerAcknowledgementCount | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0;2^20-1} |

The **TxTriggerAcknowledgementCount** parameter indicates how many trigger acknowledgement packets sent by the camera (in answer to the trigger edge packets sent before) have been received by the frame grabber.

| TxTriggerWaveformViolation | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | {0,1} |

The parameter is set to 1 by the camera operator if the operator detects a distance between two trigger edges on port TriggerI which violates the minimal edge frequency. The minimal edge frequency is 550 ns (nanoseconds) on all microEnable 5 platforms. The parameter holds its value until it has been read. After being read, the parameter updates the value. Frequency control is running permanently and is not influenced by the read status of the parameter.

## 29.38.4. Examples of Use

The use of operator CXPSingleCamera is shown in the following examples:

- Section 10.4.1, 'CoaXPress Area Scan Cameras'

  Tutorial - Basic Acquisition
- Section 10.4.1.1, 'Basic Acquisition Example for Single Line CoaXPress Area Scan Cameras'

Tutorial - Basic Acquisition

- Section 10.4.2, 'CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition

- Section 10.4.2.1, 'Basic Acquisition Example for Single Line CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition

- Section 10.5.2.1, 'Basic Acquisition Example for Single Line CoaXPress Line Scan Cameras'

  Tutorial - Basic Acquisition

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 29.39. Operator DigIOPort

**Operator Library: Hardware Platform**

This operator shows the status of four digital inputs using parameter *DigInGet*. Moreover, two digital outputs can be set using parameter *DigOutSet*. This operator represents the image data interface between a grayscale area scan GigE Vision camera and VisualApplets.

The operator exclusively occupies outputs index 3 and 7. These outputs cannot be used by other operators. Check Appendix A, '*Device Resources*' for a list of available device resources.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |

## 29.39.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |

## 29.39.2. Supported Link Format

None

## 29.39.3. Parameters

| DigInGet | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 15] |
| This parameter makes the digital input signals available to the software. Bit 0 reflects digital input0, bit 1 digital input1, bit 2 digital input2, and bit 3 digital input3. | |

| DigOutSet | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, 3] |
| This parameter makes it possible to send a signal to the digital output ports 3 and 7. Bit 0 defines the signal of the digital output 3, and bit 1 the signal of the digital output 7. | |

# 29.40. Operator DmaFromPC

**Operator Library: Hardware Platform**

**For the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platform:**

This operator provides a source of user data transferred from the user application back to the frame grabber for advanced processing. The *DmaFromPC* operator does not interpret transported data. Instead, this operator provides data in raw format 8-bit per value at parallelism degree, which corresponds to the amount of used PCIe lanes for the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms.

All data received by a DMA from the PC memory are output at the output link O. The *DmaFromPC* operator supports only one output format: 8-bit unsigned gray (raw), parallelism 32, maximal image height 1, maximal image width must not exceed 2^31-1 and must also be divisible by the parallelism 32, i.e. 2,147,483,616 pixels (bytes).

The maximal image width can be configured by the user. The setting of the output link and thus the maximal image width must be greater than or equal to the raw frame size. Otherwise, the operator injects an image into VisualApplets, which violates the link properties. Depending on the application algorithm, this image may cause issues in the processing.

The *DmaFromPC* operator requires one VisualApplets resource of type **`DmaFromHostPort`**. Set the resource index for the operator in the resource dialog. Check Section 3.8, 'Allocation of Device Resources' for more information.

**For microEnable IV VD-CL/-PoCL platforms:**

The operator DmaFromPC provides a source for image data from the PC, e.g. when the frame grabber is used as a co-processor, or well defined test images should be fed into the processing pipeline.

All data received by a DMA from the PC memory are output at the output link O. The format of the output link can be selected from 6 gray scale and color formats accessible via the parameter LinkFormat.

Because the data stream send from the PC memory does not support image dimensions, the size of the frame has to be defined using the parameters MaxNumPixel and MaxNumLines. Mind that the number of bytes transferred via DMA must match the image dimension calculated from these two parameters.

Although the parameters MaxNumPixel and MaxNumLine allow to define extremely large images there are some restrictions to observe. First of all the image size is not allowed to extend 2 Gbyte. The second restriction is that the parameter MaxNumPixel must be set in multiples of the output link parallelism.

This operator requires one VisualApplets resource of type **`DMA`**. Set the resource index for the camera in the resource dialog. Check Section 3.8, 'Allocation of Device Resources' for more information.

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |
| microEnable IV VD-CL/-PoCL |

## 29.40.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Links** | O for microEnable IV VD-CL/-PoCL, image data output<br>O for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta, image data output |

### 29.40.2. Supported Link Format

| Link Parameter | Output Link O for microEnable IV VD-CL/-PoCL | Output Link O for imaFlex CXP-12 Quad and imaFlex CXP-12 Penta |
|---|---|---|
| **Bit Width** | auto❶ | 8 |
| **Arithmetic** | {unsigned, signed} | {unsigned} |
| **Parallelism** | 4 | 32 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | {VALT_IMAGE2D} |
| **Color Format** | auto❷ | GRAY |
| **Color Flavor** | auto❸ | NONE |
| **Max. Img Width** | any | 2.147.483.616 |
| **Max. Img Height** | any | 1 |

❶ The input bit width is defined by parameter *LinkFormat*.

❷❸ The color format and color flavor are defined by the parameter *LinkFormat*.

### 29.40.3. Parameters

| **MaxNumPixel** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 1024 |
| **Range** | [1, 65536] |

This parameter defines the width of the transferred image in pixels. The value has to be less than the link property Max. Image Width.



**Availability**

This parameter is only available on the platform microEnable IV VD-CL/-PoCL, not on the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms.

| **MaxNumLines** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 1024 |
| **Range** | [1, 65536] |

This parameter defines the height of the transferred image in lines. The value has to be less than the link property Max. Image Height.



**Availability**

This parameter is only available on the platform microEnable IV VD-CL/-PoCL, not on the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms.

| **LinkFormat** | |
|---|---|
| **Type** | static parameter |
| **Default** | GRAY8x4 |
| **Range** | {GRAY8x4, GRAY16x2, GRAY32x1, RGB8x1, RGB8x2, RGB16x1} |

| **LinkFormat** | |
|---|---|
| This parameter specifies the data format of the output link O. Available formats are: | |
| GRAY8x4 | gray scale image, 8 bit per pixel, parallelism = 4 |
| GRAY16x2 | gray scale image, 16 bit per pixel, parallelism = 2 |
| GRAY32x1 | gray scale image, 32 bit per pixel, parallelism = 1 |
| RGB8x1 | RGB color image, 8 bit per pixel, parallelism = 1 |
| RGB8x2 | RGB color image, 8 bit per pixel, parallelism = 2 |
| RGB16x1 | RGB color image, 16 bit per pixel, parallelism = 1 |



**Availability**

This parameter is only available on the platform microEnable IV VD-CL/-PoCL, not on the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms.

## 29.40.4. Examples of Use

The use of operator DmaFromPC is shown in the following examples:

- Section 12.5.1, 'Co-Processor Median Filter'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a median filter is calculated.

- Section 12.5.2, 'Co-Processor Large Filter Calculation'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a large filter kernel is calculated.

# 29.41. Operator DmaToPC

**Operator Library: Hardware Platform**

The DmaToPC operator is the image interface between a VisualApplets applet and the host PC. The DMAToPC operator transfers images from the applet to the host PC. Image transfer is performed using direct memory access, i.e., the image data is directly written to the host PC memory.

The operator uses one VisualApplets resource of type **DMA**. Set the resource index for the DMA in the resource dialog. Depending on the used frame grabber, multiple DMA channels can be used. Check Section 3.8, 'Allocation of Device Resources' for more information.

Multitple color formats, bit widths and parallelisms are supported. The allowed combinations depend on the used hardware platform:

- **imaFlex CXP-12 Quad and imaFlex CXP-12 Penta:**
  - The product of the bit width and parallelism must be an integer multiple of 8 and must be less than or equal to 256.
  - The product of the maximal image width and the maximal image height must be divisible by 4 bytes.
- **microEnable 5 marathon VCLx, microEnable 5 marathon VCL, microEnable 5 marathon VCX-QP, microEnable 5 marathon VF2, and LightBridge VCL**:

  The link properties of the operator input depend on the PCIe mode that is supported by the applet. The supported PCIe mode you can select in operator AppletProperties, parameter PcieInterfaceType.

  Multiple color formats, bit widths and parallelism are supported. The maximum of the product Parallelism x Bit Width must be a multiple of 8 and depends on the PCIe mode you selected in operator AppletProperties:
  - Generation_1: Parallelism x Bit Width <= 64 Bit.
  - Generation_2: Parallelism x Bit Width <= 128 bit

  The product of the maximal image width and the maximal image height must be divisible by 4 bytes.
- **microEnable 5 ironman VD8-PoCL, microEnable 5 ironman VQ8-CXP6D (DIN connector) and microEnable 5 ironman VQ8-CXP6B (BNC connector)**:
  - When parameter **PcieInterfaceType** in the operator **AppletProperties** is set to Generation_1, the product of the bit width and the parallelism must be a multiple of 8 and less or equal to 128.
  - When parameter **PcieInterfaceType** in the operator **AppletProperties** is set to Generation_2, the product of the bit width and the parallelism must be a multiple of 8 and less or equal to 256.
- **microEnable IV VD4-CL/-PoCL and microEnable IV VQ4-GE/-GPoE**:

  For the frame grabbers using a PCIe x4 interface, the product of the bit width and parallelism must be an integer multiple of 8 and must be less than or equal to 128.
- **microEnable IV VD1-CL/-PoCL**:

  For grayscale images, the product of the parallelism and bit width must either be 16, 32, 64 or 128. In case of color formats, the product either has to be 24 or 48.



**RGB is converted to BGR**

The operator converts RGB data to BGR data (i.e., the components R and B are switched before DMA transfer). After DMA transfer, the former RGB image is available as a BGR image in the PC's RAM. This conversion is carried out to ease further processing with software tools, as most software tools directly support BGR format.

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |
| microEnable 5 marathon VCLx |
| microEnable 5 marathon VCL |
| microEnable 5 marathon VCX-QP |
| microEnable 5 marathon VF2 |
| LightBridge VCL |
| microEnable 5 ironman VQ8-CXP6D/-CXP6B |
| microEnable 5 ironman VD8-PoCL |
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |
| microEnable IV VQ4-GE/-GPoE |

### 29.41.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |

### 29.41.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | see description |
| **Arithmetic** | unsigned |
| **Parallelism** | see description |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_IMAGE2D |
| **Color Format** | {VAF_COLOR, VAF_GRAY} |
| **Color Flavor** | {FL_NONE, FL_RGB} If VAF_GRAY selected: FL_NONE; If VAF_COLOR selected: FL_RGB |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.41.3. Parameters

| CurrentTransferLength | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | |
| **Range** | [0; 2^34-1] |
| The parameter reflects the amount of byte that has already been transferred of a frame that is being processed. This information allows to analyze DMA buffer data in a software application before the entire frame is transmitted. | |

| CurrentTransferLength |
|---|
| **Availability**<br><br>mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VQ8-CXP6D, and mE5 ironman VQ8-CXP6B |

| PixelsPerLine | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | |
| The previously transferred image line length in pixels. | |
| **Availability**<br><br>microEnable IV VD4-CL/-PoCL and microEnable IV VQ4-GE/-GPoE only | |

| PixelsPerLine | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | |
| The previously transferred image line length in bytes. | |
| **Availability**<br><br>microEnable IV VD4-CL/-PoCL and microEnable IV VQ4-GE/-GPoE only | |

| ReadLongsPerLine | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | |
| This parameter provides the geometry of the transferred frame. The number of long words (4 byte) per line can be read. | |
| **Availability**<br><br>microEnable IV VD1-CL only | |

## 29.41.4. Examples of Use

The use of operator DmaToPC is shown in the following examples:

- 2. *Getting Started*

  Getting Started

- Figure 3.1, 'Simple VisualApplets Design'

  Basic Principles - Learn the Idea of VisualApplets

- Section 3.5, 'Data Flow '

  Data Flow - Learn about the Pipeline Structure used in VisualApplets

- Section 3.8, 'Allocation of Device Resources'

  Learn the allocation of the device resources of the operator.

- Section 9.2, ' Multiple DMA Channel Designs '

  Tutorial - Using multiple DMA channel outputs for one camera.

- Section 12.1.2, 'JPEG Encoder Gray'

  Examples - A simple example which shows the usage of the JPEG operators.

- Section 12.1.4, 'JPEG Compression Using Operator **JPEG_Encoder**'

  Examples - Simple examples which show the usage of the operator **JPEG_Encoder**.

- Section 12.1.5, 'JPEG Color Compression Using User Library Elements'

  Examples - Simple examples which shows the usage of the JPEG user library elements for color JPEG compression.

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.4.2.1, 'Color Plane Separation Option 1 - Three DMAs'

  Splitting the RGB color planes into three DMA channel outputs.

- Section 12.5.1, 'Co-Processor Median Filter'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a median filter is calculated.

- Section 12.5.2, 'Co-Processor Large Filter Calculation'

  Examples - The coprocessor feature of the microEnable IV VD1-CL is shown. As an example, a large filter kernel is calculated.

- Section 12.6.7, 'Image Flow Control'

  Example - For debugging purposes of the designs internal data flow control in hardware and a possible compensation.

# 29.42. Operator GPI

**Operator Library: Hardware Platform**

The operator GPI provides an interface to the digital inputs. Via parameter *Pin_ID*, you select the digital input (which is wired to a physical pin on a GPIO unit) you want to use for receiving a signal. The same digital inputs can be used by multiple operators.

For all platforms except the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms: If you use a device with several GPIO units, use parameter *ConnectorType* to select the GPIO unit you want to address (Front GPIO or GPIO). On the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms, the parameter *ConnectorType* is not available.

For the mapping of digital inputs and pin connectors, check the User Manual of your frame grabber [https://docs.baslerweb.com/frame-grabbers].

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |
| mE5 marathon VCLx |
| mE5 marathon VCL |
| mE5 marathon VCX-QP |
| mE5 marathon VF2 |
| LightBridge VCL |
| mE5 ironman VQ8-CXP6D/-CXP6B |
| mE5 ironman VD8-PoCL |
| pixelPlant 100 |
| pixelPlant 200 |

## 29.42.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, general purpose input signal to be used inside VisualApplets. |

## 29.42.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.42.3. Parameters

| PinID (imaFlex CXP-12 Quad platform and imaFlex CXP-12 Penta) | |
|---|---|
| **Type** | static write parameter |
| **Default** | FrontGpi0 |
| **Range** | {FrontGpi0, FrontGpi1, FrontGpi2, FrontGpi3, ExtensionGpi0, ExtensionGpi1, ExtensionGpi2, ExtensionGpi3, ExtensionGpi4, ExtensionGpi5, ExtensionGpi6, ExtensionGpi7} |

Via parameter *PinID*, you define the digital input (which is wired to a physical pin on a GPIO socket) you want to use for receiving a signal. The same digital inputs can be used by multiple operators.

*FrontGpiN* represents the GPI for the Front GPIO connector.

*ExtensionGpiN* represents the GPI for Extension GPIO side connector.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| Pin_ID (mE5 platforms, pixelPlant and LightBridge VCL) | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | [0;7] or [0;3] |

Via parameter *Pin_ID*, you define the digital input (which is wired to a physical pin on a GPIO socket) you want to use for receiving a signal. The same digital inputs can be used by multiple operators.

[0;3] The value range is 0-3 for the *Front GPIO* on marathon, the *Front GPIO* on LightBridge, and the *GPIO* on LightBridge.

[0;7] For the *GPIO* on marathon and in all other cases, the value range is 0-7.



**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VQ8-CXP6D/-CXP6B, mE5 ironman VD8-PoCL, pixelPlant 100, pixelPlant 200

| ConnectorType | |
|---|---|
| **Type** | static write parameter |
| **Default** | GPIO |
| **Range** | {FrontGPIO, GPIO} |

Via parameter *ConnectorType*, you define which GPIO socket (*GPIO* or *Front GPIO*) you want to address with the value you enter for parameter Pin_ID.



**Availability**

This parameter is only available for platforms that have a *Front GPIO*:

- mE5 marathon VCLx
- mE5 marathon VCL
- mE5 marathon VCX-QP
- mE5 marathon VF2

| ConnectorType |
| --- |
| • LightBridge VCL |

### 29.42.4. Examples of Use

The use of operator GPI is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.2, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Operator **TrgBoxLine**'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design. For parameters of "TrgBoxLine" on microEnable4 IV VD4-CL/PoCL platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine'.

- Section 12.15.9.2, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Operator **TrgBoxLine**'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design. For parameters of "TrgBoxLine" on microEnable IV VQ4-GE/-GPoE platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine'.

- Section 12.15.10.2, ' Line Scan Trigger for microEnable 5 marathon/LightBridge VCL with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.11.2, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.2, 'Line Scan Trigger for microEnable 5 marathon VCX QP with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.2, 'Line Scan Trigger for imaFlex CXP-12 Quad with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.2, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

# 29.43. Operator GPO

**Operator Library: Hardware Platform**

The operator GPO provides an interface to the digital outputs. Via parameter *Pin_ID*, you select the digital output (which is wired to a physical pin on a GPIO unit) you want to use for sending a signal.

For all platforms except the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms: If you use a frame grabber with two GPIO units, use parameter *ConnectorType* to select the GPIO unit you want to address (GPIO or Front GPIO). On the imaFlex CXP-12 Quad and imaFlex CXP-12 Penta platforms, the parameter *ConnectorType* is not available.

Each digital output can only be used once. Thus, the selected digital output is to be used by exclusively one instance of the GPO operator.

For each digital output, one device resource of type TriggerOut is used. See Appendix A, '*Device Resources*' for a full list of device resources.

For the mapping of digital outputs and physical pins on the GPIO units, check User Manual of your frame grabber [https://docs.baslerweb.com/frame-grabbers/index.html].

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |
| mE 5 marathon VCLx |
| mE 5 marathon VCL |
| mE 5 marathon VCX-QP |
| mE 5 marathon VF2 |
| LightBridge VCL |
| mE 5 ironman VQ8-CXP6D/-CXP6B |
| mE 5 ironman VD8-PoCL |
| pixelPlant 100 |
| pixelPlant 200 |

## 29.43.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, general purpose output signal to be send out of the frame grabber. |

## 29.43.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |

| Link Parameter | Input Link I |
|---|---|
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.43.3. Parameters

| PinID (imaFlex CXP-12 Quad and imaFlex CXP-12 Penta) | |
|---|---|
| **Type** | static write parameter |
| **Default** | FrontGpo0 |
| **Range** | imaFlex CXP-12 Penta: {FrontGpo0, FrontGpo1, FrontGpo2, FrontGpo3, ExtensionGpo0, ExtensionGpo1, ExtensionGpo2, ExtensionGpo3, ExtensionGpo4, ExtensionGpo5, ExtensionGpo6, ExtensionGpo7}<br><br>imaFlex CXP-12 Quad:{FrontGpo0, FrontGpo1, ExtensionGpo0, ExtensionGpo1, ExtensionGpo2, ExtensionGpo3, ExtensionGpo4, ExtensionGpo5, ExtensionGpo6, ExtensionGpo7} |

Via parameter *PinID*, you define the digital output (which is wired to a physical pin on a GPIO socket) you want to use for sending a signal. The same digital output can be used only once, i.e., by one instance of the GPO operator.

*FrontGpoN* represents the GPO for the Front GPIO connector.

*ExtensionGpoN* represents the GPO for Extension GPIO side connector.



**Availability**

imaFlex CXP-12 Quad, imaFlex CXP-12 Penta

| Pin_ID (mE5 platforms, pixelPlant and LightBridge VCL) | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | [0;7], [0;3] or [0;1] |

Via parameter *Pin_ID*, you define the digital ouput (which is wired to a physical pin on the GPIO or FrontGPIO socket) you want to use for sending a signal. The same digital output can be used only once, i.e., by one instance of the GPO operator.

[0;7] When you address the *GPIO* on marathon, or if you use a device with only one GPIO unit (ironman, PixelPlant), the value range is 0 - 7.

[0;3] When you address the *GPIO* on LightBridge, the value range is 0-3.

[0;1] When you address the *Front GPIO* on marathon or LightBridge, the value range is 0-1.



**Availability**

mE5 marathon VCLx, mE5 marathon VCL, mE5 marathon VCX-QP, mE5 marathon VF2, LightBridge VCL, mE5 ironman VQ8-CXP6D/-CXP6B, mE5 ironman VD8-PoCL, pixelPlant 100, pixelPlant 200

| ConnectorType | |
|---|---|
| **Type** | static write parameter |
| **Default** | GPIO |
| **Range** | {GPIO, FrontGPIO} |

**ConnectorType**

Via parameter *ConnectorType*, you define which GPIO socket (*GPIO* or *Front GPIO*) you want to address with the value you enter for parameter Pin_ID.



**Availability**

This parameter is only available for platforms that have a *Front GPIO*:

- mE5 marathon VCLx
- mE5 marathon VCL
- mE5 marathon VCX-QP
- mE5 marathon VF2
- LightBridge VCL

## 29.43.4. Examples of Use

The use of operator GPO is shown in the following examples:

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.2, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Operator **TrgBoxLine**'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design. For parameters of "TrgBoxLine" on microEnable4 IV VD4-CL/PoCL platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine'.

- Section 12.15.9.2, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Operator **TrgBoxLine**'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design. For parameters of "TrgBoxLine" on microEnable IV VQ4-GE/-GPoE platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine'.

- Section 12.15.10.2, ' Line Scan Trigger for microEnable 5 marathon/LightBridge VCL with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.11.2, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.12.2, 'Line Scan Trigger for microEnable 5 marathon VCX QP with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.13.2, 'Line Scan Trigger for imaFlex CXP-12 Quad with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.
- Section 12.15.14.2, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

# 29.44. Operator LED

**Operator Library: Hardware Platform**

The **LED** operator provides an interface for accessing the board LEDs of the frame grabber via the applet.

Driving a logic HIGH on the input forces the correspondent board LED to light on. Driving a logic LOW on the input forces the correspondent board LED to light off.

For the exact location of board LEDs on the frame grabber, see the manual of your frame grabber [https://docs.baslerweb.com/frame-grabbers/index.html].

| Available for Hardware Platforms |
|---|
| imaFlex CXP-12 Penta |
| imaFlex CXP-12 Quad |
| mE5 marathon VCLx |
| mE5 marathon VCL |
| mE5 marathon VCX-QP❶ |
| mE5 marathon VF2 |
| LightBridge VCL |
| mE5 ironman VQ8-CXP6D |
| mE5 ironman VQ8-CXP6B |
| mE5 ironman VD8-PoCL |

❶ On marathon VCX-QP, the numbering of the LEDs is mirrored, see parameter description of parameter *Port* below.

## 29.44.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, signal input |

## 29.44.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.44.3. Parameters

| Port | |
|---|---|
| **Type** | static write parameter |
| **Default** | USR1 |
| **Range** | **imaFlex CXP-12 Penta, and imaFlex CXP-12 Quad**: {USR1, USR2, USR3, USR4, USR5, USR6}<br>**LightBridge VCL, mE5 marathon VCLx, and mE5 marathon VCL**: {USR1, USR2}<br>**mE5 marathon VCX-QP and mE5 marathon VF2**: {USR1, USR2, USR3, USR4} |

The parameter defines which of the board LEDs will be connected by the operator.

**imaFlex CXP-12 Penta and imaFlex CXP-12 Quad:**

- USR1 maps the operator to the slot LED USR1.
- USR2 maps the operator to the slot LED USR2.
- USR3 maps the operator to the slot LED USR3.
- USR4 maps the operator to the slot LED USR4.
- USR5 maps the operator to the slot LED USR5.
- USR6 maps the operator to the slot LED USR6.

**mE5 marathon VCLx, mE5 marathon VCL, LightBridge VCL:**

- USR1 maps the operator to the slot LED USR1.
- USR2 maps the operator to the slot LED USR2.

**mE5 marathon VF2:**

- USR1 maps the operator to the slot LED USR1.
- USR2 maps the operator to the slot LED USR2.
- USR3 maps the operator to the slot LED USR3.
- USR4 maps the operator to the slot LED USR4.

**mE5 marathon VCX-QP:**

- USR1 maps the operator to the slot LED 4.
- USR2 maps the operator to the slot LED 3.
- USR3 maps the operator to the slot LED 2.
- USR4 maps the operator to the slot LED 1.

To turn the LED **ON**, the input I must be driven HIGH. To turn the LED **OFF**, the input I must be driven LOW.



**Platforms**

This parameter is used with the following platforms:

- imaFlex CXP-12 Penta, and imaFlex CXP-12 Quad (value range {USR1, USR2, USR3, USR4, USR5, USR6})
- mE5 marathon VCLx (value range {USR1, USR2})

| Port | |
|---|---|
| | • mE5 marathon VCL (value range {USR1, USR2})<br>• mE5 marathon VCX-QP (value range {USR1, USR2, USR3, USR4})<br>• mE5 marathon VF2 (value range {USR1, USR2, USR3, USR4})<br>• LightBridge VCL (value range {USR1, USR2}) |

| Pin_ID | |
|---|---|
| **Type** | static write parameter |
| **Default** | 0 |
| **Range** | [0;3] |

The Pin_ID parameter defines which of the 4 board LEDs will be connected by the operator. Range is [0; 3].

**Platforms**

This parameter is used with these platforms:

- mE5 ironman VQ8-CXP6D
- mE5 ironman VQ8-CXP6B
- mE5 ironman VD8-PoCL

## 29.44.4. Examples of Use

The use of operator LED is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

# 29.45. Operator NativeTrgPortIn

**Operator Library: Hardware Platform**

The operator NativeTrgportIn provides an interface to the microEnable's digital inputs. Accessible by the operator are the first four input trigger pins.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |

## 29.45.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Links** | TrgI1, signal output<br>TrgI2..TrgI4, signal output |

## 29.45.2. Supported Link Format

| Link Parameter | Output Link TrgI1 | Output Link TrgI2..TrgI4 |
|---|---|---|
| **Bit Width** | 1 | TrgI[1]] |
| **Arithmetic** | unsigned | TrgI[1] |
| **Parallelism** | 1 | TrgI[1] |
| **Kernel Columns** | 1 | TrgI[1] |
| **Kernel Rows** | 1 | TrgI[1] |
| **Img Protocol** | VALT_SIGNAL | TrgI[1] |
| **Color Format** | VAF_GRAY | TrgI[1] |
| **Color Flavor** | FL_NONE | TrgI[1] |
| **Max. Img Width** | any | TrgI[1] |
| **Max. Img Height** | any | TrgI[1] |

## 29.45.3. Parameters

None

# 29.46. Operator NativeTrgPortInExt

**Operator Library: Hardware Platform**

The operator NativeTrgportInExt provides an interface to the microEnable's digital inputs. Accessible by the operator are all eight input trigger pins.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |

## 29.46.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Links** | TrgI1, signal output<br>TrgI1..TrgI8, signal output |

## 29.46.2. Supported Link Format

| Link Parameter | Output Link TrgI1 | Output Link TrgI1..TrgI8 |
|---|---|---|
| **Bit Width** | 1 | TrgI[1]] |
| **Arithmetic** | unsigned | TrgI[1] |
| **Parallelism** | 1 | TrgI[1] |
| **Kernel Columns** | 1 | TrgI[1] |
| **Kernel Rows** | 1 | TrgI[1] |
| **Img Protocol** | VALT_SIGNAL | TrgI[1] |
| **Color Format** | VAF_GRAY | TrgI[1] |
| **Color Flavor** | FL_NONE | TrgI[1] |
| **Max. Img Width** | any | TrgI[1] |
| **Max. Img Height** | any | TrgI[1] |

## 29.46.3. Parameters

None

# 29.47. Operator NativeTrgPortOut

**Operator Library: Hardware Platform**

Provides an interface to the microEnable's digital outputs and CC outputs. The operator can access the four digital outputs and the four Camera Link trigger outputs CC1 to CC4. The operator requires one resource of type **`CameraControl`**. Set the resource index for the camera in the resource dialog. If the resource index is set to 0, the digital outputs 0 to 3 and the CC output for the camera on port A are used. If the index is set to 1, the digital outputs 4 to 7 and the CC output for camera port B are used. Check Section 3.8, 'Allocation of Device Resources' for more information.

For the mapping of digital outputs and pin connectors check the frame grabber hardware manual.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |

## 29.47.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | TrgO1..TrgO4], signal output<br>CC1..CC4, signal output |

## 29.47.2. Supported Link Format

| Link Parameter | Input Link TrgO1..TrgO4] | Input Link CC1..CC4 |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

## 29.47.3. Parameters

None

# 29.48. Operator RxLink

**Operator Library: Hardware Platform**

This operator provides a data link between the frame grabber to the pixelPlant boards Px100 and Px200 respectively from the pixelPlant boards to the frame grabbers. TxLink represents the input interface operator. The link is capable to transport data in any image format. The format of the link has to be parameterized to exactly meet with the corresponding link which is sending the data.

The parameter *Channel_ID* specifies the unique ID of the virtual data channel. This is necessary to address the corresponding sender and establish the channel communication. The channel ID occupies one device resource of type RxLink. Check Appendix A, '*Device Resources*' for a full list of device resources.

Data transfers are controlled by flow control of VisualApplets and do not need buffering, e.g. a RxLink input can be directly connected to a DmaFromPc operator without utilizing ImageBuffer operators. However if an infinite source is used like any of the camera operators, buffering is still required before TxLink.

The data link is a virtual channel between the px100/200 boards and the mE4VD4-CL board. The maximal number of virtual RxLink channels must not exceed 61. Also consider to use as less as possible links to use the bandwidth and FPGA resources effectively. All virtual RxLink data channels are mapped internally on a single physical link of 1GByte/s bandwidth.

The Channel_IDs can be in any order and can start from any index between 1 and 61. However all indices must be unique. Note that on the sender side on the TxLinks must have the same IDs as the RxLinks on the receiver side.

TxLinks and RxLink on the same board do not share the same physical channel and are fully independent, i.e. RxLink and TxLink operators on the same board can have the same or different Channel_IDs and are not related to each other in any way.



**Optimized Routing of designs with PixelPlant**

To optimize the routing results (during build) of designs for mE4 with PixelPlant PX100/ PX200/PX200e, we recommend to use settings for bit width and parallelism that ensure that the product of bit width and parallelism is a multiple of 64. The optimum routing results can be expected if the product is exactly 64.

product = n * 64

n = 1 leads to optimal routing results.

Very good routing results can also be expected if the product of bit width and parallelism is a power of two and less or equal 64, but not 1, 2, or 4. Other configurations may lead to timing errors.

| Available for Hardware Platforms |
|---|
| mmicroEnable IV VD4-CL/-PoCL |
| pixelPlant 100 |
| pixelPlant 200 |

## 29.48.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, signal input |

### 29.48.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | [1, 64] |
| **Arithmetic** | {unsigned, signed} |
| **Parallelism** | any |
| **Kernel Columns** | any |
| **Kernel Rows** | any |
| **Img Protocol** | {VALT_SIGNAL, VALT_LINE1D, VALD_PIXEL0D} |
| **Color Format** | any |
| **Color Flavor** | any |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.48.3. Parameters

| Channel_ID | |
|---|---|
| **Type** | static parameter |
| **Default** | 1 |
| **Range** | [1, 61] |
| This parameter defines the unique channel ID for the data link. | |

### 29.48.4. Examples of Use

The use of operator RxLink is shown in the following examples:

- Section 4.6, 'PixelPlant Designs'

  Generating PixelPlant design and establishing interconnections.

# 29.49. Operator TrgPortArea

**Operator Library: Hardware Platform**

This operator generates the trigger (Exsync) for the Camera. It is possible to use an internal signal generator (GrabberControlled) or to trigger the signal generator from external signals, either by input signals from the trigger expansion board (TTL Trigger board or OPTO Trigger board), or by software.

The signal generator can produce Exsync signals with a flexible delay, pulse width and polarity. Additionally a Flash signal, which is available at the trigger expansion board, is generated.

The operator occupies 4 digital outputs and one camera port for the CC signals. Hence, one VisualApplets resource of type **`CameraControl`** is used. If the device resource is set to 0, the digital outputs 0 to 3 and camera port A is used. However, if the resource index is set to 1, the digital outputs 4 to 7 and camera port B is used. Note that no other operators can use these hardware resources. Set the resource index in the resource dialog. Check Section 3.8, 'Allocation of Device Resources' for more information

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.49.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |

## 29.49.2. Supported Link Format

None

## 29.49.3. Parameters

| TriggerMode | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | GrabberControlled |
| **Range** | {GrabberControlled, ExternSw_Trigger} |
| This parameter selects the operation mode for the internal Exsync signal generator. The source for the external trigger input can by selected via the parameter ImgTrgInSource.<br><br>GrabberControlled: Exsync is generated periodically by the internal signal generator.<br><br>ExternSw_Trigger: An external trigger signal is used to start the signal generator once. | |

| ExsyncEnable | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | ON |
| **Range** | {OFF, ON} |
| Enables or disables the Exsync output to the camera. | |

| ExsyncFramesPerSec | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 8 Hz |
| **Range** | depends on parameter Accuracy |

| ExsyncFramesPerSec |
|---|
| This parameter specifies the frequency of the Exsync generation. |

| ExsyncExposure | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 4000 µs |
| **Range** | depends on parameter Accuracy |
| This parameter specifies the pulse width of the Exsync signal, which can be used by many cameras to specify the exposure time. Therefore, it is possible to adjust the exposure time via software, even while grabbing. Enter the value in microseconds. | |

| ExsyncDelay | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 µs |
| **Range** | depends on parameter Accuracy |
| This parameter specifies the delay of the generated Exsync signal, with respect to an external trigger input. Therefore, it is possible to synchronize the exposure interval to a flash. Enter the value in microseconds. | |

| ExsyncPolarity | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter adjusts the polarity of the Exsync signal generator the polarity accepted by the connected camera. Use LowActive, if the camera opens the shutter on a falling edge, otherwise use HighActive. | |

| ImgTrgInSource | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | InSignal0 |
| **Range** | {InSignal0, InSignal1, InSignal2, InSignal3, InSignal4, InSignal5, InSignal6, InSignal7} |
| This parameter specifies the signal source, which is used to trigger the Exsync signal generator. This is only relevant if the TriggerMode is set to ExternSw_Trigger. | |

| ImgTrgInPolarity | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter defines the polarity of the external input trigger signal. When set to LowActive, the Exsync generator starts on a falling edge of the signal specified by the parameter ImgTrgInSource. Otherwise, the Exsync generation starts on a rising edge. This is only relevant if the TriggerMode is set to ExternSw_Trigger. | |

| ImgTrgDownscale | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 1 |
| **Range** | {1, 65535} |
| This value defines a downscaling factor for incoming trigger pulses. Only every Nth trigger puls is accepted, with number N defined by this parameter. For example, if ImgTrgDownscale is set to 3, then every 3rd incoming trigger puls will be accepted, i.e. the first two trigger pulses are discarded. | |

| FlashEnable | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | OFF |
| **Range** | {OFF, ON} |
| Enables or disables the flash output. The pulse width of the flash signal is equal to the Exsync period. | |

| FlashPolarity | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter defines the polarity for the generated Flash signal. | |

| FlashDelay | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 µs |
| **Range** | {0, 10230} with stepsize 10 µs |
| This parameter specifies the delay of the generated Flash signal, with respect to an external trigger input. Therefore, it is possible to synchronize the flash to the external trigger input. Enter the value in microseconds. | |

| SoftwareTrgPulse | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 1 |
| **Range** | {1} |
| Setting this parameter to 1, will generate a software trigger. This is only relevant if the TriggerMode is set to ExternSw_Trigger and ImgTrgInSource is set to SoftwareTrigger. | |

| SoftwareTrgDeadTime | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 10000 µs |
| **Range** | depends on parameter Accuracy |
| This parameter specifies the minimal time interval between software trigger pulses. Enter the value in microseconds. | |

| SoftwareTrgIsBusy | |
|---|---|
| **Type** | dynamic/static read parameter |
| **Default** | 0 |
| **Range** | {0, 1} |
| The SoftwareTrgIsBusy parameter enables software readout of the busy state. If busy then this parameter is set to 1 to reflect an ongoing image capture. If set to 0 then the operator is not busy. | |

| CC1output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Hdsync, HdsyncInvert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC1 line of the CameraLink cable. | |

| CC2output | |
|---|---|
| **Type** | dynamic/static write parameter |

| CC2output | |
|---|---|
| **Default** | Vcc |
| **Range** | {Exsync, ExsyncInvert, Hdsync, HdsyncInvert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC2 line of the CameraLink cable. | |

| CC3output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Vcc |
| **Range** | {Exsync, ExsyncInvert, Hdsync, HdsyncInvert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC3 line of the CameraLink cable. | |

| CC4output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Vcc |
| **Range** | {Exsync, ExsyncInvert, Hdsync, HdsyncInvert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC4 line of the CameraLink cable. | |

| Accuracy | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 10 µs |
| **Range** | [0.05, 500] µs |
| This parameter specifies the time base for the trigger system. It defines the jitter and the ranges for delays, periods etc. Enter the value in microseconds. | |

| DebouncingMaxTime | |
|---|---|
| **Type** | static write parameter |
| **Default** | 65.520 us |
| **Range** | [0.016, 1000000] us |
| This parameter specifies the maximum limit of debouncing time. Enter the value in microseconds. | |

| DebouncingTime | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0.112 us |
| **Range** | [0.016, 1000000] us |
| This parameter specifies the debouncing time. The input trigger signals must keep the same value to be detected as such. Fast signal changes within the debounce time will be filtered out. Enter the value in microseconds. | |

## 29.49.4. Examples of Use

The use of operator TrgPortArea is shown in the following examples:

- Section 12.15.1.2, ' Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Operator **TrgPortArea**'

  An area scan trigger using the operator **TrgPortArea** is presented. Please read for information on this operator Section 29.49, 'TrgPortArea'.

# 29.50. Operator TrgPortLine

**Operator Library: Hardware Platform**

This operator generates the trigger (Exsync) for the camera. It is also responsible for assembling the acquired lines to images.

It is possible to use an internal signal generator (GrabberControlled) or to trigger the signal generator from external signals, either by input signals from the trigger expansion board (TTL Trigger board or OPTO Trigger board), or by software. The signal generator can produce Exsync signals with a flexible delay, pulse width and polarity. Additionally, a flash signal which is available at the trigger expansion board is generated.

The operator occupies 4 digital outputs and one camera port for the CC signals. Hence, one VisualApplets resource of type **`CameraControl`** is used. If the resource index is set to 0, the digital outputs 0 to 2 and camera port A is used. If the resource index is set to 1, the digital outputs 4 to 6 and camera port B is used. Outputs 3 and 7 are not used by the operator. Note that no other operators can use these hardware resources. Set the resource index in the resource dialog.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |

## 29.50.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 29.50.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_LINE1D | VALT_IMAGE2D |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | 65536 | as I |

## 29.50.3. Parameters

| YOffset | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, $2^{24}$] |
| This parameter defines the number of lines omitted at the beginning of a frame. | |

| YLength | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [8, $2^{24}$] |
| This parameter defines the number of lines of a frame. | |

| MaxGatedHeight | |
|---|---|
| **Type** | dynamid read/write parameter |
| **Default** | restricted |
| **Range** | {restricted, unrestricted} |

The parameter *MaxGatedHeight* allows you to limit the maximum image height when the image trigger mode (*ImgTriggerMode*) is set to ExternSw_Gate.

If if the image trigger mode (*ImgTriggerMode*) is set to ExternSw_Gate, and the parameter *MaxGatedHeight* is set to unrestricted, the image height is defined by the time the gate is open, i.e., by the pulse width of the external image trigger signal or the duration of the software trigger being value 1. Now, if the gate is open for a long time, the image height gets large. If parameter *MaxGatedHeight* is set to restricted, the image height is limited to YLength image lines even if the gate is still open. The operator will discard any further lines and wait for the next open gate to start a new frame.

In contrast, if the parameter is set to unrestricted, the image height is only defined by the gate.



**Violation of Link Property Possible**

Using this parameter in unrestricted mode can cause a violation of the VisualApplets link protocol. The image height could exceed the maximum allowed image height defined in the output link of the TrgPortLine operator. Be careful when using the unrestricted mode. See Section 3.7.2, 'Link Properties' for more information on link properties.

A successive SplitImage operator can divide large images into chunks (smaller images).

| LineTriggerMode | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | GrabberControlled |
| **Range** | {GrabberControlled, Extern_Trigger, GrabberControlled_Gated_by_Img, Extern_Trigger_Gated_by_Img} |

This parameter selects the operation mode for the internal Exsync signal generator. The source for the external trigger input can by selected via the parameters LineTrgInSourceA and LineTrgInSourceB (see below).

GrabberControlled: Exsync is generated periodically by the internal signal generator

Extern_Trigger: An external trigger signal is used to start the signal generator once

GrabberControlled_Gated_by_Img: Exsync is generated periodically by the internal signal generator during the acquisition of a frame

Extern_Trigger_Gated_by_Img: An external trigger signal is used to trigger the signal generator during the acquisition of a frame

| ExsyncEnable | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | OFF |

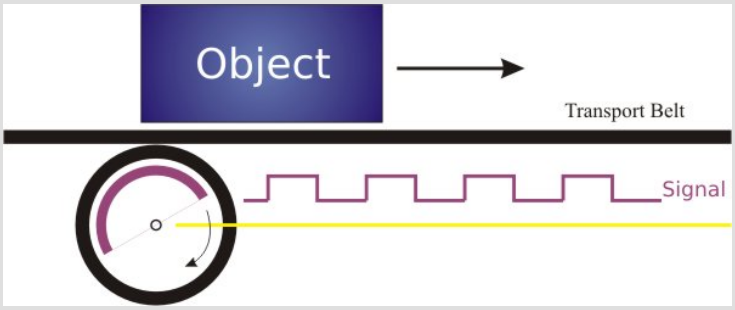
| ExsyncEnable | |
|---|---|
| **Range** | {OFF, ON} |
| Enables or disables the Exsync output to the camera. | |

| LineTrgInSourceA | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | InSignal0 |
| **Range** | {InSignal0, InSignal1, InSignal2, InSignal3, InSignal4, InSignal5, InSignal6, InSignal7} |
| This parameter specifies the signal source which is used to trigger the Exsync signal generator. This is only relevant if the TriggerMode is set to Extern_Trigger. | |

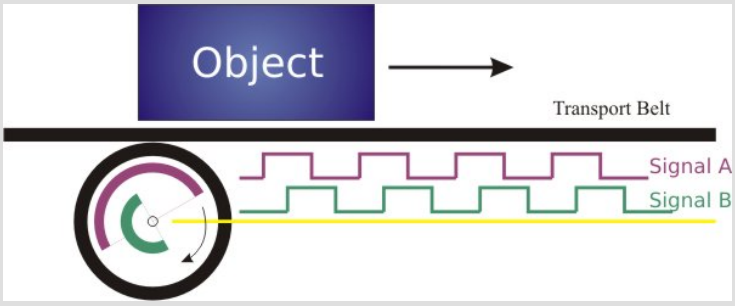
| LineTrgInSourceB | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | InSignal0 |
| **Range** | {InSignal0, InSignal1, InSignal2, InSignal3, InSignal4, InSignal5, InSignal6, InSignal7} |
| This parameter specifies the signal source which is used to trigger the Exsync signal generator. This is only relevant if the TriggerMode is set to Extern_Trigger and EncoderABMode is set to Signal_AB_Filter. | |

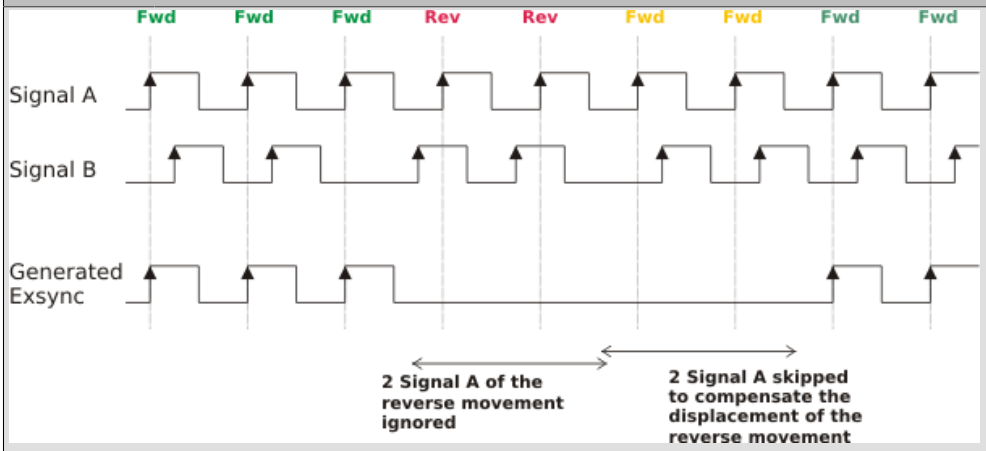
| EncoderABMode | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | Signal_A_Only |
| **Range** | {Signal_A_Only, Signal_AB_Filter, Signal_ABx2_Filter, Signal_ABx4_Filter} |

This parameter specifies whether a single trigger input (A only) is used for the Exsync generation, or the signals A and B.




Signal A/B support enables to determine the revolving direction of the shaft encoder and to suppress and compensate backward movements:



| EncoderABMode |
|---|
|  |
| Signal_A_Only: The trigger input selected by LineTrgInSourceA is used for Exsync generation.<br><br>Signal_AB_Filter: Exsync is generated for a forward rotation of the shaft encoder in single resolution, i.e., a trigger pulse for a rising edge of LineTrgInSourceA.<br><br>Signal_ABx2_Filter: Exsync is generated for a forward rotation of the shaft encoder in double resolution, i.e., a trigger pulse for a rising edge of LineTrgInSourceA and a falling edge of LineTrgInSourceA. Both edges of LineTrgInSourceA are used.<br><br>Signal_ABx4_Filter: Exsync is generated for a forward rotation of the shaft encoder in quad resolution, i.e., a trigger pulse for a rising and a falling edge of LineTrgInSourceA and a rising and a falling edge of LineTrgInSourceB.<br><br>Related Parameters when AB support enabled:<br><br>• *EncoderABLead* (possibility to switch the definition of forward)<br><br>You can reset the shaft encoder by setting parameter *EncoderABMode* to value *Signal_A_Only*. |

| EncoderABLead | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | Signal_AB |
| **Range** | {Signal_AB, Signal_BA} |
| A foreward movement is defined by a rising edge of signal A before signal B if the parameter is set to Signal_AB, or vice versa:<br><br>Signal_AB: Forward is defined by A before B<br><br>Signal_BA: Forward is defined by B before A | |

| LineTrgInPolarity | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter defines the polarity of the external input trigger signal LineTrgInSourceA and LineTrgInSourceB. When set to LowActive, the Exsync generator starts on a falling edge of the signal specified by the parameter ImgTrgInSource. Otherwise, the Exsync generation starts on a rising edge. This is only relevant if the TriggerMode is set to Extern_Trigger. | |

| LineTrgDownscaler | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |

| LineTrgDownscaler | |
|---|---|
| **Range** | [1, 256] |
| This parameter specifies the number of external input trigger signals, which are needed to generate the Exsync. This is only relevant if the TriggerMode is set to an external trigger mode. | |

| LineTrgPhase | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | [1, 256] |
| This parameter specifies the number of external input trigger signals, which are needed to generate the first Exsync of a frame. This is only relevant if the TriggerMode is set to Extern_Trigger_Gated_by_Img. | |

| ExsyncPeriod | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 100 µs |
| **Range** | [1.024, 4000] µs |
| This parameter specifies the period of the Exsync signal. Therefore, it defines the line frequency when using the grabber controlled mode to trigger the connected camera. | |

| ExsyncExposure | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 20 µs |
| **Range** | [1.024, 2000] µs, must not exceed ExsyncPeriod |
| This parameter specifies the pulse width of the Exsync signal, which can be used by many cameras to specify the exposure time. Therefore, it is possible to adjust the exposure time via software, even while grabbing. | |

| Exsync2Delay | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 µs |
| **Range** | [0, 2000] µs, must not exceed ExsyncPeriod |
| This parameter specifies the delay of the generated Exsync signal, with respect to an external trigger input. Therefore, the Exsync2 signal is a delayed clone of the Exsync (polarity, period, etc. are the same as for Exsync). | |

| ExsyncPolarity | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter adjusts the polarity of the Exsync signal genarator to the polarity accepted by the connected camera. Use LowActive, if the camera opens the shutter on a falling edge, otherwise use HighActive. | |

| ImgTriggerMode | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | FreeRun |
| **Range** | {FreeRun, ExternSw_Trigger, ExternSw_Gate} |
| This parameter selects the operation mode for the internal Image Gate. | |

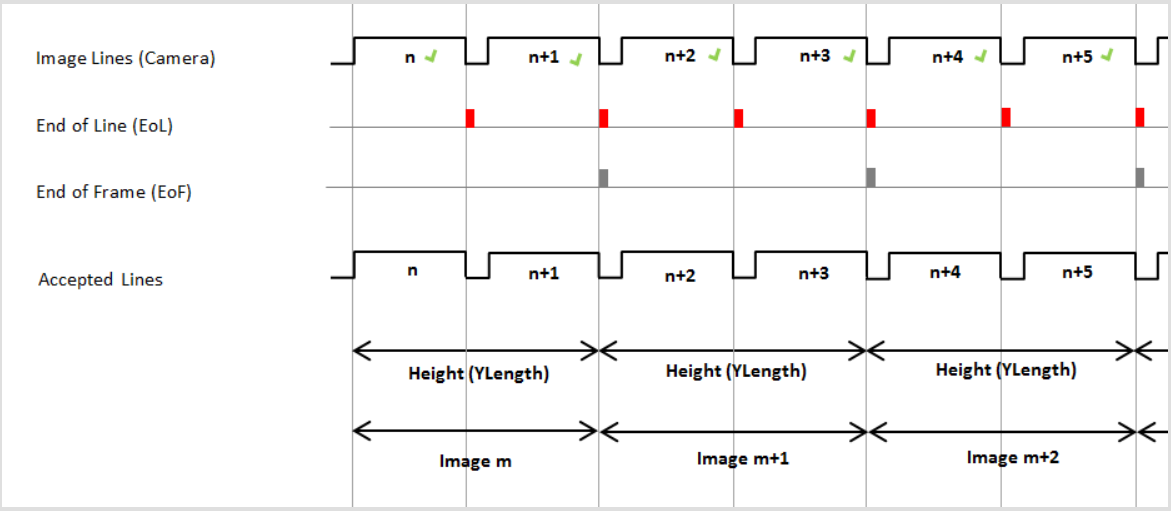
**ImgTriggerMode**

The image trigger input signal may be created by external (peripheral) devices (e.g., shaft encoder), or by software. The source for the external image trigger input you can select via the parameter *ImgTrgInSource*, see below).

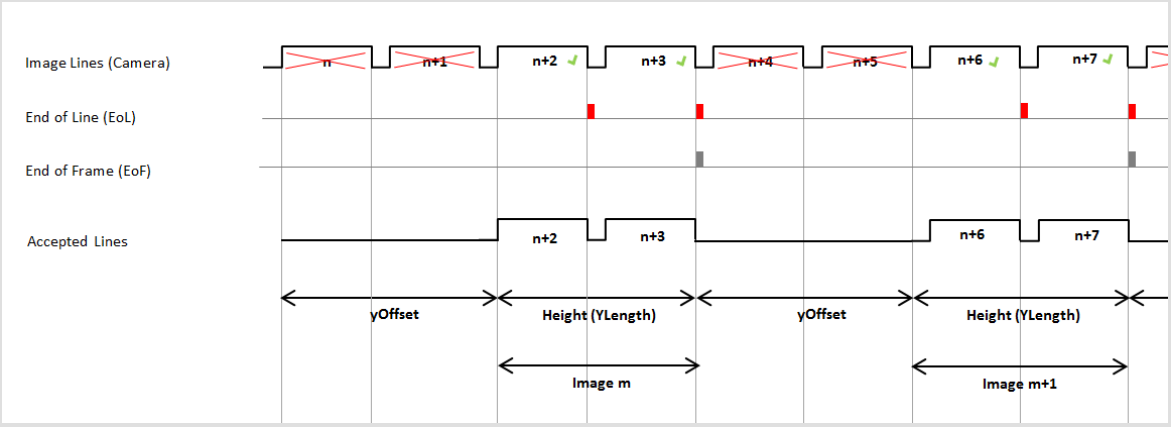
The values of parameter ImgTriggerMode induce the following behaviour:

- **FreeRun**:
  - All incoming lines transmitted by the camera are accepted.
  - The Image Gate is ignored.
  - Parameter *YLength* defines when the end of frame (EOF) is generated.
  - If parameter *yOffset* is greater than Zero, a number of yOffset lines between two sequential images is ommitted.

    **Example**: *yOffset* = 0, *YLength* = 2:




    **Example**: *yOffset* = 2, *YLength* = 2:

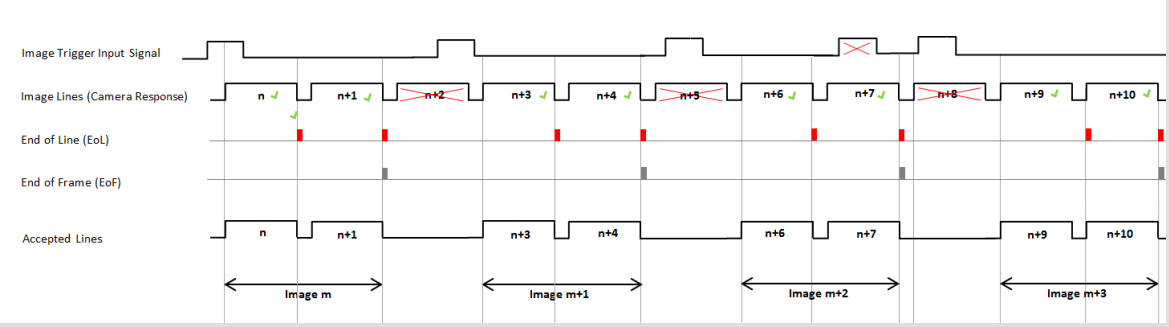



- **ExternSw_Trigger**:
  - At the rising edge of an accepted image trigger input signal, a new frame is started and incoming lines are appended to an image up to *YLength*.

**ImgTriggerMode**

- Parameter *YLength* defines when the end of frame (EOF) is generated.
- A new rising edge of the image trigger is only accepted after *YLength* lines have been appended and EOF is generated. A new rising edge of the image trigger is ignored if it occurs before *YLength* lines have been appended to an image and EOF is generated.

  **Example**: *YLength* is set to value 2:



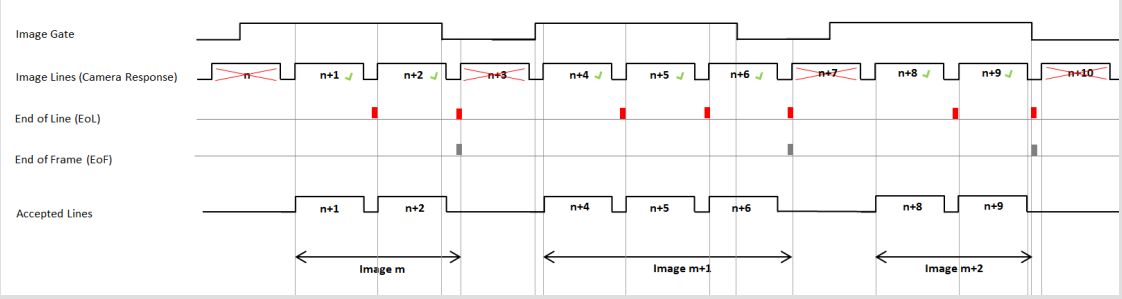

- **ExternSw_Gate**:
  - In gated mode, it is important how long the image trigger input signal is active, since it functions as the Image Gate signal.
  - The exact behaviour of this mode can be defined by the parameter *MaxGatedHeight*. (With *MaxGatedHeight* you can define if the maximum image height of the images is restricted or unrestricted. For details, see description of parameter *MaxGatedHeight*.)

    **Behaviour of mode ExternSW_Gate when parameter *MaxGatedHeight* is set to value "Unrestricted"**

    - While the Image Gate signal is active, the incoming lines are appended to an image.
    - The Image Gate controls which image lines are valid:
      - While the Image Gate signal is active, the incoming lines are accepted. (See figure, e.g., lines n+8 and n+9.)
      - While the Image Gate signal is not active, the incoming lines are ignored. (See figure, e.g., line n+3.)
      - Each line which starts while the Image Gate signal is active is valid. (See figure: n+1, n+2, n+4, n+5, n+6, n+8 and n+9 are valid.)

        If the falling edge of the Image Gate occurs before all the data of the last line are acquired, the line is nevertheless valid and acquired completely. (See figure: n+2 and n+6)
      - There are always complete lines being transferred.
      - Each line which starts while the Image Gate is down is not valid. (See figure: Line n and n +7 are not valid, line n+4 is valid.)

| ImgTriggerMode |
|---|




- End of line (EoL) and end of frame (EoF) are either generated directly before the start of a new line, or at the falling edge of the Image Gate.

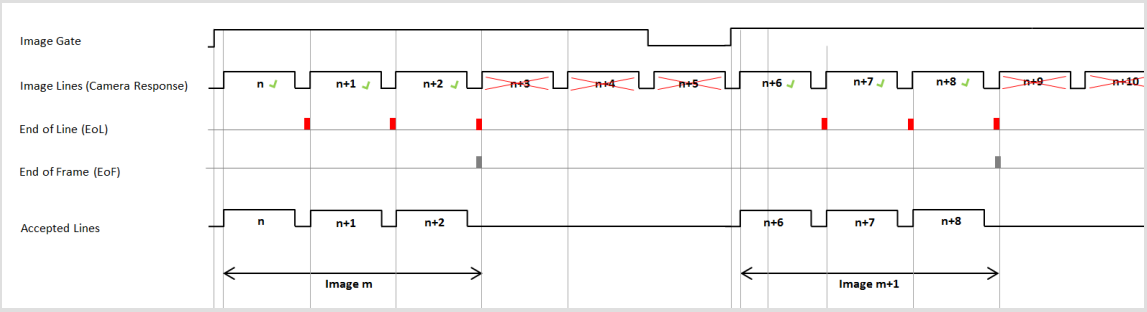
  *EoL directly before start of new line*: If the falling edge of a line occurs within the open Image Gate and the Image Gate covers the start of the next line, the EoL is generated directly before the rising edge of the next line. (See figure: n+1, n+4, n+5 and n+8.)

  *EoL together with EoF directly before start of new line*: If the falling edge of the Image Gate occurs before the falling edge of a line, EoL and EoF are generated directly before the rising edge of the next line. (See figure: Image m and image m+1.)

  *EoL together with EoF at falling edge of Image Gate*: If the falling edge of a line occurs within the open Image Gate and the falling edge of the Image Gate occurs before the rising edge of the next line, EoL and EoF are generated directly after the falling edge of the Image Gate. (See figure: Image m+2.)

**Behaviour of Mode ExternSW_Gate when parameter *MaxGatedHeight* is set to value "Restricted"**
If parameter *MaxGatedHeight* is set to "restricted", the behaviour is the same as described above for *MaxGatedHeight* with value "unrestricted", with the following exceptions:

- As long as the image gate is open, the lines are appended up to a maximal image height of*YLength*. All following lines are ommitted until the next rising edge of the Image Gate signal.
- If a falling edge of the Image Gate signal occurs before YLength lines are appended: The falling edge of the Image Gate signal defines the height of the image, even if the image has not the height of YLength. (See parameter *MaxGatedHeight* for more information about the image height.)
- The example below (figure) shows the behaviour at *MaxGatedHeight* = restricted and *YLength* = 3.




| ImgTrgInSource | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | InSignal0 |

| ImgTrgInSource | |
|---|---|
| **Range** | {InSignal0, InSignal1, InSignal2, InSignal3, InSignal4, InSignal5, InSignal6, InSignal7} |
| This parameter specifies the signal source which is used to trigger the image acquisition. This is only relevant if the ImgTriggerMode is set to ExternSw_Trigger or ExternSw_Gate. | |

| ImgTrgInPolarity | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter defines the polarity of the external input trigger signal. | |

| ImgTrgDelay | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 0 |
| **Range** | {0, 4095} |
| The parameter delays the image trigger signal by the given number of image lines. | |

| FlashEnable | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | OFF |
| **Range** | {OFF, ON} |
| Enables or disables the flash output. The pulse width of the flash signal is equal to one line period. | |

| FlashPolarity | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter defines the polarity for the generated Flash signal. | |

| FlashDelay | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | {0, 4095} |
| This parameter specifies the number of lines to delay the generated Flash signal, with respect to an external trigger input. Therefore, it is possible to synchronize the flash to the external trigger input.<br><br>The pulse width of the flash signal is equal to one line period. | |

| SoftwareTrgPulse | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | |
| **Range** | {1} |
| Setting this parameter to 1 will generate a software trigger. This is only relevant if the TriggerMode is set to an external trigger mode and ImgTrgInSource is set to SoftwareTrigger. | |

| SoftwareTrgInput | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | |

| SoftwareTrgInput | |
|---|---|
| **Range** | {0, 1} |
| With this parameter a software gate can be produced for the image trigger mode ExternSw_Gate. | |

| ImgTrgIsBusy | |
|---|---|
| **Type** | dynamic/static read parameter |
| **Default** | 0 |
| **Range** | {0, 1} |
| The ImgTrgIsBusy parameter enables software readout of the busy state for the image trigger. If busy then this parameter is set to 1 to reflect an ongoing image capture. If set to 0 then the operator is not busy. | |

| CC1output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Hdsync, HdsyncInvert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC1 line of the CameraLink cable. | |

| CC2output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Hdsync, HdsyncInvert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC2 line of the CameraLink cable. | |

| CC3output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Hdsync, HdsyncInvert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC3 line of the CameraLink cable. | |

| CC4output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Hdsync, HdsyncInvert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC4 line of the CameraLink cable. | |

| ImgTrgDebouncingMaxTime | |
|---|---|
| **Type** | static write parameter |
| **Default** | 65.520 us |
| **Range** | [0.016, 1000000] us |
| This parameter specifies the maximal time for ImgTrgDebouncingTime parameter. The smaller the maximal time the less FPGA resources are required to implement the debouncing timer. | |

| ImgTrgDebouncingTime | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 0.112 us |
| **Range** | [0.016, ImgTrgDebouncingMaxTime] us |
| This parameter specifies the debouncing time the input image trigger signal must keep the same value to be detected as such. Fast signal changes within the debounce time will be filtered out. | |

| LineTrgDebouncingMaxTime | |
|---|---|
| **Type** | static write parameter |
| **Default** | 65.520 us |
| **Range** | [0.016, 1000000] us |
| This parameter specifies the maximal time for LineTrgDebouncingTime parameter. The smaller the maximal time the less FPGA resources are required to implement the debouncing timer. | |

| LineTrgDebouncingTime | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 0.112 us |
| **Range** | [0.016, ImgTrgDebouncingMaxTime] us |
| This parameter specifies the debouncing time the input line trigger signals must keep the same value to be detected as such. Fast signal changes within the debounce time will be filtered out. | |

## 29.50.4. Examples of Use

The use of operator TrgPortLine is shown in the following examples:

- Section 3.8, 'Allocation of Device Resources'

  Learn the allocation of the device resources of the operator.

- Section 10.1.2, 'Camera Link Line Scan Cameras'

  Tutorial - Basic Acquisition

- Section 10.3.2, 'Camera Link Line Scan Cameras '

  Tutorial - Basic Acquisition

- Section 12.8.4.4, 'Filter for Line Scan Cameras'

  Examples - Explains how to implement a filter for line scan cameras.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

# 29.51. Operator TriggerIn

**Operator Library: Hardware Platform**

The operator TriggerIn provides an interface to the microEnable's digital inputs. By using parameter *Pin_ID*, the digital input is selected.

For the mapping of digital inputs and pin connectors check the frame grabber hardware manual.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |
| microEnable IV VQ4-GE/-GPoE |

## 29.51.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, signal output |

## 29.51.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.51.3. Parameters

| Pin_ID | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 7] |
| This parameter defines which digital input is used. The same inputs can be used by multiple operators. | |

## 29.51.4. Examples of Use

The use of operator TriggerIn is shown in the following examples:

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 29.52. Operator TriggerOut

**Operator Library: Hardware Platform**

The operator TriggerIn provides an interface to the microEnable's digital outputs. By using parameter *Pin_ID*, the digital outputs is selected.

Each output can only be used once. Hence, the selected output has to be exclusively used by this operator and one device resource of type TriggerOut is used. See Appendix A, '*Device Resources*' for a full list of device resources.

For the mapping of digital outputs and pin connectors check the frame grabber hardware manual.

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |
| microEnable IV VQ4-GE/-GPoE |

## 29.52.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, signal input |

## 29.52.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.52.3. Parameters

| Pin_ID | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 7] |
| This parameter defines which digital output is used. | |

## 29.52.4. Examples of Use

The use of operator TriggerOut is shown in the following examples:

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 29.53. Operator TxLink

**Operator Library: Hardware Platform**

This operator provides a data link between the frame grabber to the pixelPlant boards Px100 and Px200 respectively from the pixelPlant boards to the frame grabbers. TxLink represents the output interface operator. The link is capable to transport data in any image format. The format of the link is specified automatically by the operator connected to the TxLink operator.

The parameter *Channel_ID* specifies the unique ID of the virtual data channel. This is necessary to address the corresponding receiver and establish the channel communication. The channel ID occupies one device resource of type TxLink. Check Appendix A, '*Device Resources*' for a full list of device resources.

Data transfers are controlled by flow control of VisualApplets and do not need buffering, e.g., an RxLink input can be directly connected to a DmaFromPc operator without utilizing ImageBuffer operators. However, if an infinite source is used like any of the camera operators, buffering is still required before TxLink.

The data link is a virtual channel between the px100/200 boards and the mE4VD4-CL board. The maximal number of virtual TxLink channels must not exceed 61. Also consider to use as less as possible links to use the bandwidth and FPGA resources effectively. All virtual TxLink data channels are mapped internally on a single physical link of 1GByte/s bandwidth.

The Channel_IDs can be in any order and can start from any index between 1 and 61. However all indices must be unique. Note that on the receiver side on the RxLinks must have the same IDs as the TxLinks on the sender side.

TxLinks and RxLink on the same board do not share the same physical channel and are fully independent, i.e. RxLink and TxLink operators on the same board can have the same or different Channel_IDs and are not related to each other in any way.



**Optimized Routing of designs with PixelPlant**

To optimize the routing results (during build) of designs for mE4 with PixelPlant PX100/ PX200/PX200e, we recommend to use settings for bit width and parallelism that ensure that the product of bit width and parallelism is a multiple of 64. The optimum routing results can be expected if the product is exactly 64.

product = n * 64

n = 1 leads to optimal routing results.

Very good routing results can also be expected if the product of bit width and parallelism is a power of two and less or equal 64, but not 1, 2, or 4. Other configurations may lead to timing errors.

| Available for Hardware Platforms |
|---|
| mmicroEnable IV VD4-CL/-PoCL |
| pixelPlant 100 |
| pixelPlant 200 |

## 29.53.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, signal input |

### 29.53.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | [1, 64] |
| **Arithmetic** | {unsigned, signed} |
| **Parallelism** | any |
| **Kernel Columns** | any |
| **Kernel Rows** | any |
| **Img Protocol** | {VALT_SIGNAL, VALT_LINE1D, VALD_PIXEL0D} |
| **Color Format** | any |
| **Color Flavor** | any |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 29.53.3. Parameters

| Channel_ID | |
|---|---|
| **Type** | static parameter |
| **Default** | 1 |
| **Range** | [1, 61] |
| This parameter defines the unique channel ID for the data link. | |

### 29.53.4. Examples of Use

The use of operator TxLink is shown in the following examples:

- Section 4.6, 'PixelPlant Designs'

  Generating PixelPlant design and establishing interconnections.

# 29.54. Operator SignalToEvent

**Operator Library: Hardware Platform**

The operator generates software events for rising edges at its input links.

All N inputs (up to 16) can generate individual events. The event will not provide any signal link data. Commonly, this operator is used to monitor the status of GPIOs or to signal special conditions.

For each input link, an event name must be specified. This is done using parameter *EventNameInput[n]*. The event name will then be the complete module name plus the event name e.g. Device1_Process0_HierarchicalBoxName_SignalToEventModuleName_ValueOfParameterEventNAmeInput[n]. All events can be accessed with the Framegrabber API functions for events. Check the Framegrabber API Manual [https://docs.baslerweb.com/frame-grabbers/framegrabber-api.html] for more information.

The operator will automatically allocate N Event and one EventSource device resources. Check Appendix A, '*Device Resources*' for more information.

| Available for Hardware Platforms |
|---|
| microEnable IV VD4-CL/-PoCL |
| microEnable IV VQ4-GE/-GPoE |

## 29.54.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I0..In, signal input |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.

## 29.54.2. Supported Link Format

| Link Parameter | Input Link I0..In |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 29.54.3. Parameters

| EventsWithTimestamp | |
|---|---|
| **Type** | static/dynamic read/write parameter |

| EventsWithTimestamp | |
|---|---|
| **Default** | ON |
| **Range** | {ON, OFF} |
| Defines if high-precision timestamps are attached to each event. ON = timestamps are generated. OFF = no timestamps are generated. | |

| EventNameInput[n] | |
|---|---|
| **Type** | static write parameter |
| **Default** | EventNameInput[n] |
| **Range** | |
| Every event input must be assigned a unique identifier name. This event name is used to identify and use a particular hardware event signal in the Framegrabber SDK. | |

# 30. Library Prototype




Disclaimer: The *Prototype* library is absolutely preliminary and subject to extensive changes, even removal. The functionality of the operators cannot be guaranteed. Therefore, the operators must not be used for productive designs. It is included for testing purposes only.

The following list summarizes all Operators of Library Prototype

| Operator Name | | Short Description | available since |
|---|---|---|---|
| CNT | COUNTER | Hardware Multiplier. | Version 1.1 |
| EDIF | CustomSignalOperator | Hardware Multiplier. | Version 1.1 |
| * | HWMULT | Hardware Multiplier. | Version 1.1 |
| RLE | PackbitsRLE | Simple image compression through run-length encoding (RLE). | Version 1.1 |
| LINE | TrgBoxLine | Generates the trigger (Exsync signal) for the Camera and is also responsible to assemble the acquired lines to images. | Version 1.2 |
| XYZ | RGB2XYZ | Converts the color space from RGB to XYZ. | Version 1.1 |
| | XYZ2LAB | Converts the color space from XYZ to LAB. | Version 1.1 |

Table 30.1. Operators of Library Prototype

# 30.1. Operator COUNTER

**Operator Library: Prototype**



**Warning**

Disclaimer: This module is part of the prototype library. It is absolutely preliminary and subject to extensive changes, even removal. Therefore, this module must not be used for productive designs. It is included for testing purposes only.

See the following code to understand how the operator works:

```
if (Clr == 1) {
      O = 0;
    } else if (Add == 1) {
      O += I;
    }
    return O;
```

## 30.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I1, data input<br>Add, conditon for the addition<br>Clr, conditon to clear |
| **Output Link** | O, data output |

## 30.1.2. Supported Link Format

| Link Parameter | Input Link I1 | Input Link Add |
|---|---|---|
| **Bit Width** | [1, 64] | 1 |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Clr | Output Link O |
|---|---|---|
| **Bit Width** | 1 | [1, 64] |
| **Arithmetic** | as I | as I |
| **Parallelism** | as I | as I |
| **Kernel Columns** | 1 | any |
| **Kernel Rows** | 1 | any |
| **Img Protocol** | as I | as I |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |

| Link Parameter | Input Link Clr | Output Link O |
|---|---|---|
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 30.1.3. Parameters

| ClearEol | |
|---|---|
| **Type** | static/dynamic write parameter |
| **Default** | 0, dynamic |
| **Range** | [0, 1] |
| This parameter an additional condition for clearing (setting to zero) the counter. If this parameter is 1, the counter is cleared at each end of line. | |

| ClearEof | |
|---|---|
| **Type** | static/dynamic write parameter |
| **Default** | 0, dynamic |
| **Range** | [0, 1] |
| This parameter an additional condition for clearing (setting to zero) the counter. If this parameter is 1, the counter is cleared at each end of frame. | |

## 30.2. Operator CustomSignalOperator

**Operator Library: Prototype**



**Warning**

Disclaimer: This module is part of the prototype library. It is absolutely preliminary and subject to extensive changes, even removal. Therefore, this module must not be used for productive designs. It is included for testing purposes only.

The operator imports a custom netlist in the Electronic Design Interchange Format (EDIF) which processes on signal data. To configure behaviour of the custom operator a configuration interface with write and read access is included. For proper work in Visual Applets a latency by the use of Latency parameter must be specified. The minimal value for latency is 2 because inputs and outputs of the operator are registered. Each pipeline stage in the custom operator from the input to the output increases operator latency. Also it is inevitable that the netlist defines the following I/O ports.

| I/0 | Signal Width | Description |
|---|---|---|
| ClkI | 1 | 62.5 MHz on mE4 platforms, 125 MHz on mE5 platforms |
| EnableI, ResetI | 1 | Used for correct behaviour in Visual Applets |
| DataI | N | Input array of signal data with N 1-bit input links |
| DataO | 1 | Output of processed signal data |
| ConfigAddressI | 8 | address asserted by parameter WriteAddress and ReadAddress to access internal configuration space |
| ConfigWriteDataI | 32 | data for configuration space |
| ConfigWriteStrobeI | 1 | asserted for one ClkI cycle after parameter WriteAddress was accessed |
| ConfigReadDataO | 32 | data from configuration space |
| ConfigReadStrobeI | 1 | asserted for one ClkI cycle after parameter ReadAddress was accessed |

### 30.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I[n], signal data input, N is in range from 1 to 64 |
| **Output Link** | O, signal data output |

### 30.2.2. Supported Link Format

| Link Parameter | Input Link I[n] | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |

| Link Parameter | Input Link I[n] | Output Link O |
|---|---|---|
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 30.2.3. Parameters

| EdifFileName | |
|---|---|
| **Type** | static write parameter |
| **Default** | netlist.edn |
| **Range** | |
| This parameter specifies the name of the netlist for this operator. Netlist file needs to be in {Visual Applets installation dir}/cores/Spartan3 path | |

| Latency | |
|---|---|
| **Type** | static write parameter |
| **Default** | 2 |
| **Range** | [2, 255] |
| This parameter specifies the latency of signals from input to ouput. All signals should have exact this latency. Because inputs and outputs of the operator are registered the default latency is 2. | |

| ConfigWriteData | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^32-1] |
| Applies value to ConfigWriteDataI. | |

| ConfigReadData | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^32-1] |
| Data at ConfigReadDataO. | |

| ConfigWriteAddress | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^8-1] |
| Applies value to ConfigAddressI and asserts ConfigWriteStrobeI for one ClkI cycle. | |

| ConfigReadAddress | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | 0 |
| **Range** | [0, 2^8-1] |
| Applies value to ConfigAddressI and asserts ConfigWriteStrobeI for one ClkI cycle. | |

# 30.3. Operator HWMULT

**Operator Library: Prototype**



**Warning**

Disclaimer: This module is part of the prototype library. It is absolutely preliminary and subject to extensive changes, even removal. Therefore, this module must not be used for productive designs. It is included for testing purposes only.

The module HWMULT multiplies the input link I1 by the input link I2 and the result I1*I2 is available at the output link. It provides identical functionality to the operator Mult. The operator uses dedicated multiplier elements in the FPGA if thoses resources are available.

## 30.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I1, data input<br>I2, data input |
| **Output Link** | O, data output of I1*I2 |

## 30.3.2. Supported Link Format

| Link Parameter | Input Link I1 | Input Link I2 | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 16] | [1, 16] | auto |
| **Arithmetic** | {unsigned, signed} | as I1 | as I1 |
| **Parallelism** | any | as I1 | as I1 |
| **Kernel Columns** | any | as I1 | as I1 |
| **Kernel Rows** | any | as I1 | as I1 |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | as I1 | as I1 |
| **Color Format** | VAF_GRAY | as I1 | as I1 |
| **Color Flavor** | FL_NONE | as I1 | as I1 |
| **Max. Img Width** | any | as I1 | as I1 |
| **Max. Img Height** | any | as I1 | as I1 |

## 30.3.3. Parameters

None

## 30.3.4. Examples of Use

The use of operator HWMULT is shown in the following examples:

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend' Tutorial - From equation to implementation. Explanation on how to implement the overlay blend.
- Section 12.1.7, 'Laser Triangulation'

Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.5, '1D Shading Correction Using Block RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in block RAM memory.

- Section 12.14.6, '1D Shading Correction Using Frame Grabber RAM'

  Examples - The example shows an 1D shading correction. The correction values are stored in Frame Grabber RAM.

# 30.4. Operator PackbitsRLE

**Operator Library: Prototype**



**Disclaimer**

Library: Prototype

This module is part of the Prototype library. It is in absolutely preliminary state and subject to extensive changes, even removal. Therefore, this module must not be used for productive designs. It is included for testing purposes only.

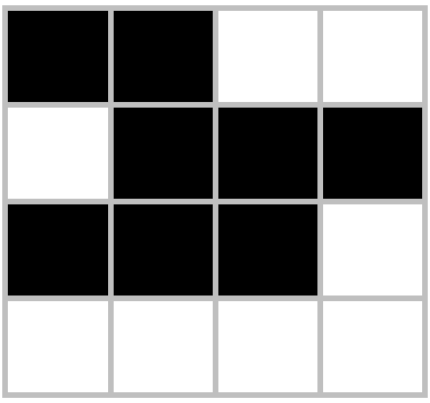
The operator PackbitsRLE performs a run-length encoding compression (RLE) of the image stream at input link I, using the packbits algorithm. Packbits was first introduced by Apple and is one of the RLE compression algorithms that can be used for lossless compression of TIFF-files.

**The Packbits Algorithm**

A raster image file consists of a header (containing information such as image dimension, color depth, etc.) and data that describe the color of each pixel. The data are written sequentially, i.e., the pixel color information has to be interpreted as pixel rows from left to right, top to bottom.

In uncompressed mode, color information for n adjacent pixels of the same color is written n times. This causes redundancy.

Example:



The figure represents a 4-pixel by 4-pixel, black-and-white image. In uncompressed format, the pixels are represented as follows (B=Black, W=White): BBWWWBBBBBBWWWWW This redundancy is reduced by the packbits compression.

The packbits data stream consists of packets with a one-byte header followed by data. The header is a signed byte.

The data describe each pixel with one byte. If the header contains a value n {-127 to -1}, the byte following the header is repeated -n+1 times in the decompressed output. Thus, redundant information (identical pixel color) is stored once together with a counter (header). For example, BBBBBB would be stored as -6B. For the first pixel of another color, a new header is generated.

If the header contains a value n {0 to 127}, n+1 bytes following the header are left uncompressed (copied literally).

The longest pixel series that can be compressed consists of 128 bytes.

After compression with packbits, the example image above (BBWWWBBBBBBWWWWW) becomes -2B-3W-6B-5W, i.e. can be represented by 8 symbols (bytes) instead of 16.



**Effectiveness**

Packbits is used for gray scale, palette, and bitonal images. The greater the number of consecutive pixels of identical color, the more effectively the operator can compress the data. Best compression results are achieved for bitonal (e.g., black-and-white) images.

The advantage of the (pretty simple) packbits algorithm lies in its speed and its extremely little need of memory (256 bytes) and processing power.

### 30.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 30.4.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [4, 12] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I❶ |
| **Max. Img Height** | any | as I |

❶ The output image width must not exceed 2^31 - 1.

### 30.4.3. Parameters

None

### 30.4.4. Examples of Use

The use of operator PackbitsRLE is shown in the following examples:

- Section 12.1.9, 'Packbits Run Length Encoder'

  Examples - A packbits run length encoding example

## 30.5. Operator TrgBoxLine

**Operator Library: Prototype**

This operator generates the trigger (Exsync signal) for the Camera. It is also responsible for assembling the acquired lines to images.

**Parameterization of Time Values**

This operator is optimized for use with microEnable 5 products (mE5 VD8-PoCL, mE5 VQ8-CXP6B, mE5 VQ8-CXP6D, LightBridge 2 VCL, marathon VCL, marathon VCLx). If you want to use this operator with microEnable IV frame grabbers, you have to double all parameter values that are messured in time units (e.g., µs).

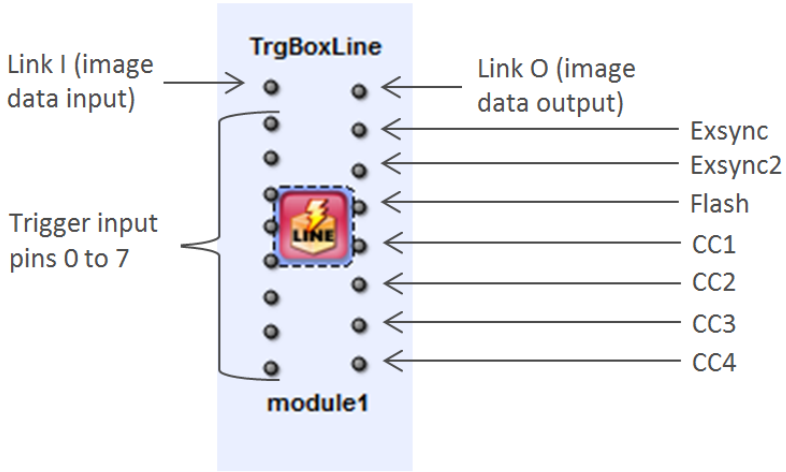
In a design, this operator is placed directly after the camera operator, since it communicates with the camera and triggers the actual image data acquisition. On the other hand, it receives image data from the camera (on data input link I) and forwards these image data (with lines already assembled to images) on data output link O to the next operator(s).

You can either use an internally produced signal (mode GrabberControlled), or external signals to activate the operator's trigger signal generator. The external signals the operator can receive come either from software (software trigger), or from peripheral devices (via slot bracket trigger port or trigger expansion board (TTL or OPTO trigger board)). The operator's trigger signal generator can produce

- Exsync signals
- Exsync signals with a flexible delay, pulse width and polarity
- a Flash signal

The Flash is triggered by an external trigger input (like,e.g., the Image Gate). However, the Flash has its own delay (in line ticks). The Flash signal lasts for one line tick and might be most useful when synchonizing additional frame grabbers or LightBridge devices.

The operator offers 8 trigger input and 7 trigger output ports. The input ports you can connect to the signal sources you want to use (e.g., software trigger, peripheral devices like shaft encoder via individual trigger board pins, other frame grabber boards via flatband cable, etc.). The output ports you can connect to the signal receivers that need to be triggered (e.g., Exsync and Exsync2 for the camera(s) on the (master) frame grabber, Flash for synchronizing further frame grabbers, etc.)




The CC signals can be used for Exsync, Exsync2 and Flash signals (also inverted), or for Vcc/Gnd. If you don't need to use all CC signals, you can connect the unnecessary CC ports to a trash operator instance.

Some of the dynamic parameters of this operator cannot be re-set during image acquisition. As soon as image acquisition is put to a halt, the parameters are accessible again. In microDisplay, the according input fields are disabled. When working with the Framegrabber API, you need to know which parameters are concerned.

The values for the following parameters cannot be re-set during image acquisition:

- ImgTriggerMode
- MaxGatedHeight
- LineTriggerMode
- YOffset
- YLength

| Available for Hardware Platforms |
|---|
| microEnable IV VD1-CL/-PoCL |
| microEnable IV VD4-CL/-PoCL |
| microEnable IV VQ4-GE/GPoE |
| microEnable 5 VD8-PoCL |
| microEnable 5 VQ8-CXP6(B) |
| microEnable 5 VQ8-CXP6D |
| LightBridge VCL |

### 30.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, data input<br>Trigger Input Links (0-7), trigger input |
| **Output Links** | O, data output<br>Trigger Output Links (0-6), trigger output |

### 30.5.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_LINE1D | VALT_IMAGE2D |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | 65536 | as I |

| Link Parameter | Input Link Trigger Input Links (0-7) | Output Link Trigger Output Links (0-6) |
|---|---|---|
| **Bit Width** | [1] | [1] |

| Link Parameter | Input Link Trigger Input Links (0-7) | Output Link Trigger Output Links (0-6) |
|---|---|---|
| **Arithmetic** | {unsigned} | {unsigned} |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 30.5.3. Parameters

| **YOffset** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | [0, $2^{24}$] |
| This parameter defines the number of lines omitted at the beginning of a frame. | |

| **YLength** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [8, $2^{24}$] |
| This parameter defines the number of lines of a frame if parameter *ImgTriggerMode* is set to ExternSw_Gate and parameter *MaxGatedHeight* is set to restricted. | |

| **MaxGatedHeight** | |
|---|---|
| **Type** | dynamid read/write parameter |
| **Default** | restricted |
| **Range** | {restricted, unrestricted} |

The parameter *MaxGatedHeight* allows you to limit the maximum image height if parameter *ImgTriggerMode* is set to ExternSw_Gate.

If parameter *ImgTriggerMode* is set to ExternSw_Gate, and parameter *MaxGatedHeight* is set to unrestricted, the image height is defined by the time the gate is open, i.e., by the pulse width of the external image trigger signal or the duration of the software trigger being value 1.

>If the gate is open for a long time, the image height gets large. If parameter *MaxGatedHeight* is set to restricted, the image height is limited to *YLength* image lines (as specified with parameter *YLength*) even if the gate is still open. The operator will discard any further lines and wait for the next open gate to start a new frame.

In contrast, if the parameter is set to unrestricted, the image height is only defined by the gate.



**Violation of Link Property Possible**

Using this parameter in unrestricted mode can cause a violation of the VisualApplets link protocol. The image height could exceed the maximum allowed image height defined in the output link of the operator. Be careful when using the unrestricted mode. See Section 3.7.2, 'Link Properties' for more information on link properties.

| MaxGatedHeight | |
|---|---|
| | A successive SplitImage operator can divide large images into chunks (smaller images). |

| LineTriggerMode | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | GrabberControlled |
| **Range** | {GrabberControlled, Extern_Trigger, GrabberControlled_Gated_by_Img, Extern_Trigger_Gated_by_Img} |

This parameter selects the operation mode for the internal Exsync signal generator. The source for the external trigger input can by selected via the parameters LineTrgInSourceA and LineTrgInSourceB (see below).

GrabberControlled: Exsync is generated periodically by the internal signal generator.

Extern_Trigger: An external trigger signal (software trigger or trigger signal from peripheral device) is used to start the signal generator once.

GrabberControlled_Gated_by_Img: Exsync is generated periodically by the internal signal generator during the acquisition of a frame.

Extern_Trigger_Gated_by_Img: An external trigger signal (software trigger or trigger signal from peripheral device) is used to trigger the signal generator during the acquisition of a frame.

| ExsyncEnable | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | OFF |
| **Range** | {OFF, ON} |

Enables or disables the Exsync output to the camera.

| LineTrgInSourceA | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | InSignal0 |
| **Range** | {InSignal0, InSignal1, InSignal2, InSignal3, InSignal4, InSignal5, InSignal6, InSignal7} |

This parameter specifies the signal source which is used to trigger the Exsync signal generator. This is only relevant if the TriggerMode is set to Extern_Trigger.

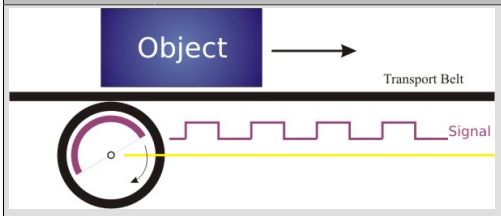
| LineTrgInSourceB | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | InSignal0 |
| **Range** | {InSignal0, InSignal1, InSignal2, InSignal3, InSignal4, InSignal5, InSignal6, InSignal7} |

This parameter specifies the signal source which is used to trigger the Exsync signal generator. This is only relevant if the TriggerMode is set to Extern_Trigger and EncoderABMode is set to Signal_AB_Filter.

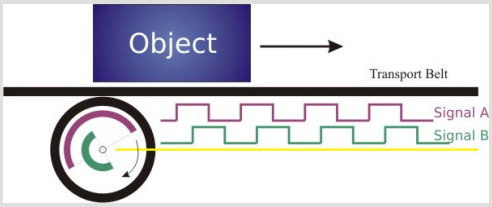
| EncoderABMode | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | Signal_A_Only |
| **Range** | {Signal_A_Only, Signal_AB_Filter, Signal_ABx2_Filter, Signal_ABx4_Filter} |

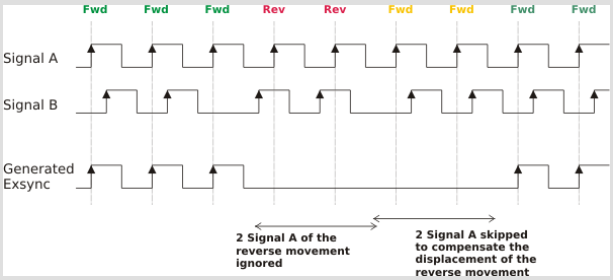
This parameter specifies whether a single trigger input (A only) is used for the Exsync generation, or the signals A and B.

| **EncoderABMode** |
|---|




Signal A/B support enables to determine the revolving direction of the shaft encoder and to suppress and compensate backward movements:




Signal_A_Only: The trigger input selected by LineTrgInSourceA is used for Exsync generation.

Signal_AB_Filter: Exsync is generated for a forward rotation of the shaft encoder in single resolution, i.e., a trigger pulse for a rising edge of LineTrgInSourceA.

Signal_ABx2_Filter: Exsync is generated for a forward rotation of the shaft encoder in double resolution, i.e., a trigger pulse for a rising edge of LineTrgInSourceA and a falling edge of LineTrgInSourceA. Both edges of LineTrgInSourceA are used.

Signal_ABx4_Filter: Exsync is generated for a forward rotation of the shaft encoder in quad resolution, i.e., a trigger pulse for a rising and a falling edge of LineTrgInSourceA and a rising and a falling edge of LineTrgInSourceB.

Related Parameters when AB support enabled:

- *EncoderABLead* (possibility to switch the definition of forward)
- *EncoderCompensation* (possibility to set backward movement compensation to on or off)
- *EncoderCompensationCount* (possibility to define an offset of encoder steps, i.e., to suppress the trigger signal output of the operator for a defined number of encoder steps)

You can reset the shaft encoder and parameter *EncoderCompensationCount* by setting parameter *EncoderABMode* to value *Signal_A_Only*.

| **EncoderABLead** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | Signal_AB |
| **Range** | {Signal_AB, Signal_BA} |

| **EncoderABLead** |
|---|
| A foreward movement is defined by a rising edge of signal A before signal B if the parameter is set to Signal_AB, or vice versa:<br><br>Signal_AB: Forward is defined by A before B<br><br>Signal_BA: Forward is defined by B before A |

| **EncoderCompensation** | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | ON |
| **Range** | {ON/OFF} |

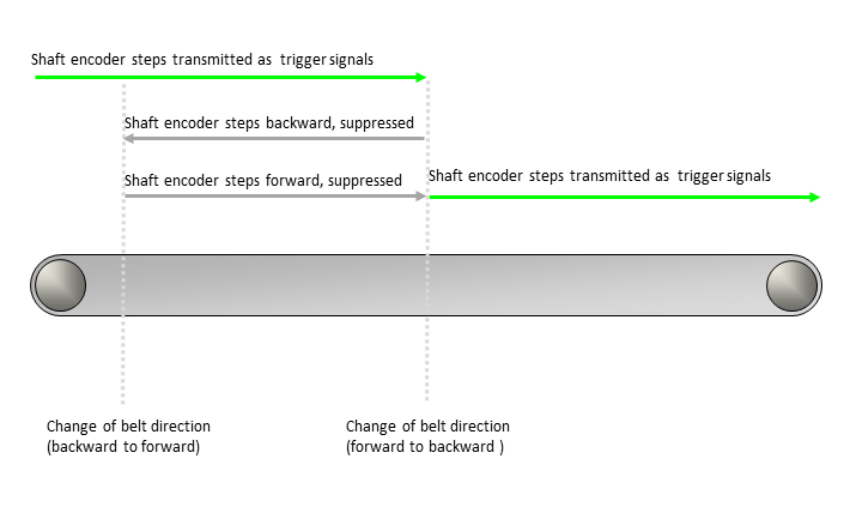
With parameter *EncoderCompensation* you can switch the compensation of the shaft encoder backward movement to ON or OFF.

This parameter is only relevant if parameter *EncoderABMode* is set to A/B support.

**ON**

If switched to ON, in case of shaft encoder backward movement the operator counts how many shaft encoder steps the shaft encoder moves backwards. When the shaft encoder moves forwards again, this number of shaft encoder steps (now forward direction) is not transmitted as external trigger signals. Only after the transportation belt is back to the place where the backward movement started, the shaft encoder steps (forward direction) are transmitted as external trigger signals again.

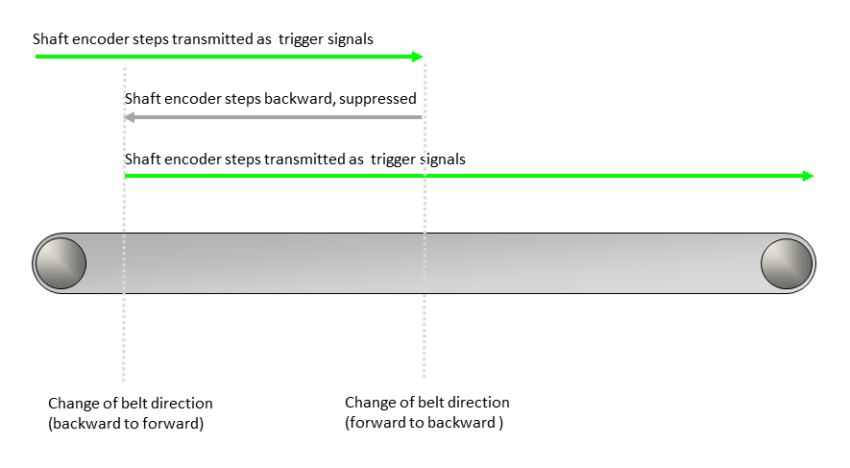
Parameter *EncoderCompensation* = ON:




**OFF**

If switched to OFF, the operator simply doesn't transmit any trigger signals as long as the transportation belt moves backwards. As soon as the transport belt starts to move forwards again, the operator transmits the shaft encoder steps (forward direction) as trigger signals.

Parameter *EncoderCompensation* = OFF:

| EncoderCompensation |
|---|
|  |
| By setting parameter *EncoderCompensation* to OFF, you reset parameter *EncoderCompensationCount* to value *0*. |

| EncoderCompensationCountBits | |
|---|---|
| **Type** | static write parameter |
| **Default** | 20 bit |
| **Range** | {8 bit ... 31 bit} |
| This parameter allows to define the value range of parameter *EncoderCompensationCount*. | |

| EncoderCompensationCount | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | $2^{20}$ -1 |
| **Range** | {0 ... $2^{\text{EncoderCompensationCountBits}}$ -1} (unit: shaft encoder steps) |

This parameter is only relevant if parameter *EncoderCompensation* is set to ON and parameter *EncoderABMode* is set to A/B support.

This parameter allows to define an offset for the forward movement of the transportation belt. This is especially helpful if the transportation belt stopps and/or moves backwards.

A specific value range for parameter *EncoderCompensationCount* you can define with parameter *EncoderCompensationCountBits*.

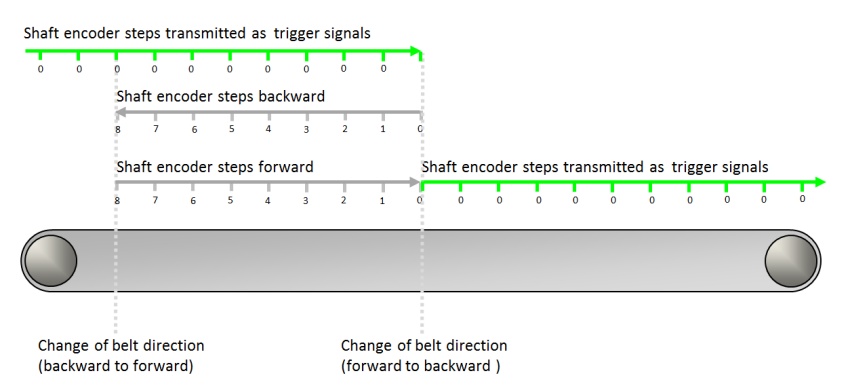
You can reset the shaft encoder and parameter *EncoderCompensationCount* by setting the parameter *EncoderABMode* to value *Signal_A_Only*.

Alternativlely, you can reset parameter *EncoderCompensationCount* to value *0* by setting parameter *EncoderCompensation* to OFF.

**Basic Conditions**

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

**EncoderCompensationCount**




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

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

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

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

**Reading the Parameter**

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

**Defining an Offset**

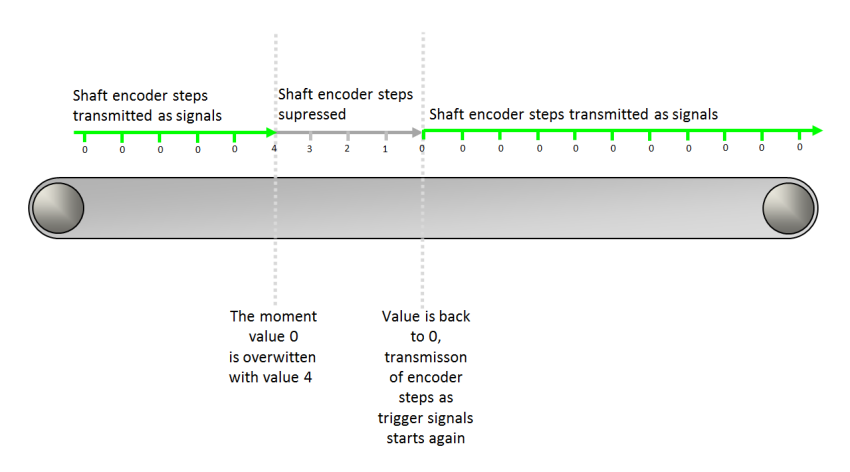
On the other hand, you can always modify the parameter value since you have write access during acquisition. If you need to define an offset to the standard encoder compensation, you can use this parameter to enter the number of steps you need the offset to be.

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

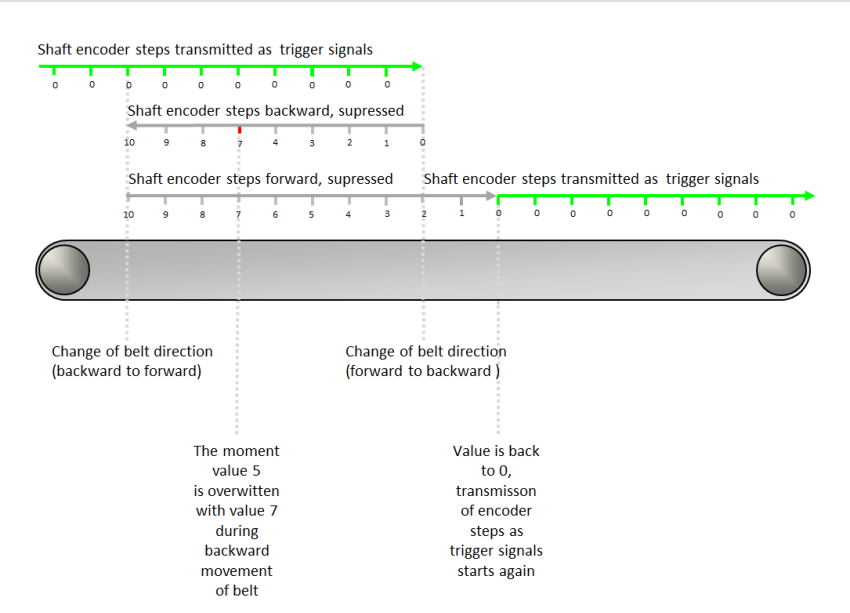
Let's look at some examples for overwriting the current value of *EncoderCompensationCount*:

**Example 1:** The transportation belt is moving forward, the shaft encoder steps are transmitted as trigger signals, and the value of EncoderCompensationCount is 0. Then, the value 0 of EncoderCompensationCount is overwritten by value 4. Result: 4 shaft encoder steps are not transmitted as trigger signals.

**EncoderCompensationCount**

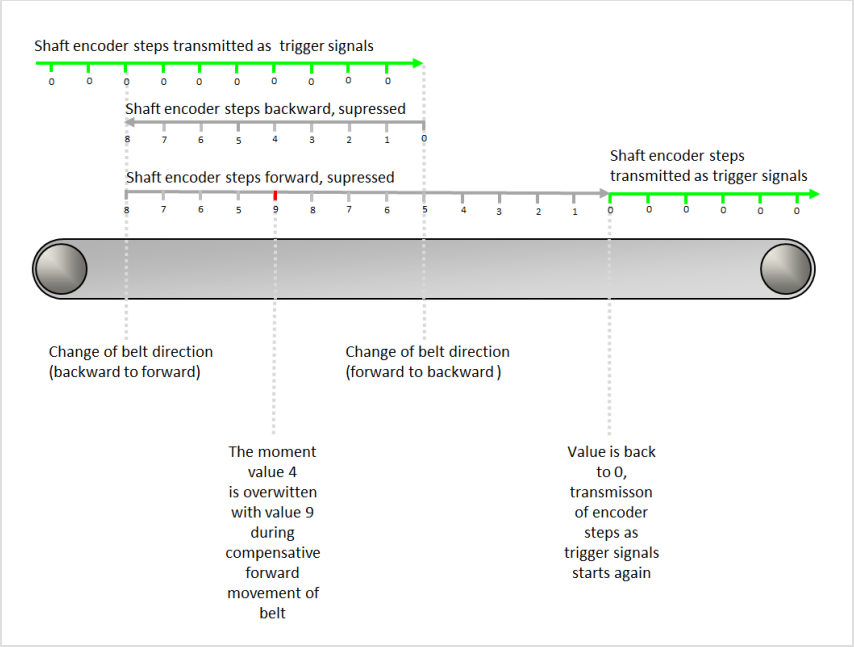



**Example 2:** The transportation belt is moving backward, the (backward) shaft encoder steps are supressed, and the value of EncoderCompensationCount is ≠0. Then, during backward movement of the transportation belt, the value 5 of EncoderCompensationCount is overwritten by value 7. Result: Offset of 2 shaft encoder steps.

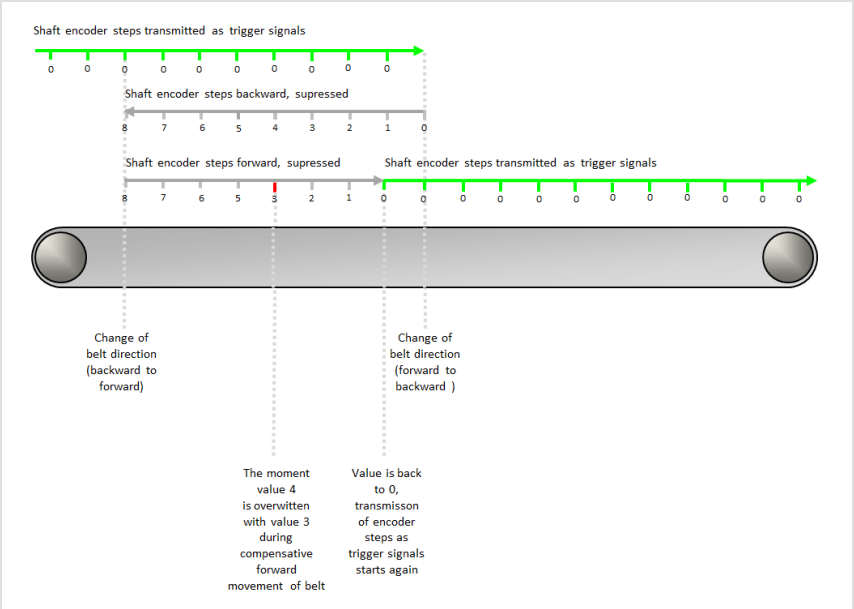



**Example 3:** The transportation belt is moving forward during compensation, the (forward) shaft encoder steps are supressed, and the value of EncoderCompensationCount is ≠0. Then, during compensative forward movement of the transportation belt, the value 4 of EncoderCompensationCount is overwritten with value 9. Result: Offset of 5 shaft encoder steps.

| EncoderCompensationCount |
|---|




**Example 4:** The transportation belt is moving forward during compensation, the (forward) shaft encoder steps are supressed, and the value of EncoderCompensationCount is ≠0. Then, during compensative forward movement of the transportation belt, the value 4 of EncoderCompensationCount is overwritten with a **smaller** value, in our case with value 3. Result: Negative offset of -1 shaft encoder step.




| LineTrgInPolarity | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |

The parameter defines the polarity of the external input trigger signal LineTrgInSourceA and LineTrgInSourceB. When set to LowActive, the Exsync generator starts on a falling edge of the

| LineTrgInPolarity |
|---|
| signal specified by the parameter ImgTrgInSource. Otherwise, the Exsync generation starts on a rising edge. This is only relevant if the TriggerMode is set to Extern_Trigger. |

| LineTrgDownscaler | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | [1, 256] |
| This parameter specifies the number of external input trigger signals, which are needed to generate the Exsync. This is only relevant if the TriggerMode is set to an external trigger mode. | |

| LineTrgPhase | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | [1, 256] |
| This parameter specifies the number of external input trigger signals, which are needed to generate the first Exsync of a frame. This is only relevant if the TriggerMode is set to Extern_Trigger_Gated_by_Img. | |

| ExsyncPeriod | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 100 µs |
| **Range** | [1.024, 2097.14] |

This parameter specifies the period of the Exsync signal. Therefore, it defines the line frequency when using the grabber controlled mode to trigger the connected camera.



**Parameterization of Time Values**

Operator *TrgBoxLine* is optimized for use with microEnable 5 products. If you work with microEnable IV frame grabbers, you have to double all parameter values that are messured in time units (e.g., µs).

| ExsyncExposure | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 20 µs |
| **Range** | [1.024, 2000] µs, must not exceed ExsyncPeriod |

This parameter specifies the pulse width of the Exsync signal, which can be used by many cameras to specify the exposure time. Therefore, it is possible to adjust the exposure time via software, even while grabbing.



**Parameterization of Time Values**

Operator *TrgBoxLine* is optimized for use with microEnable 5 products. If you work with microEnable IV frame grabbers, you have to double all parameter values that are messured in time units (e.g., µs).

| Exsync2Delay | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 µs |
| **Range** | [0, ExsyncPeriod] µs, must not exceed ExsyncPeriod |

| Exsync2Delay |
|---|
| This parameter specifies the delay of the generated Exsync signal, with respect to an external trigger input. Therefore, the Exsync2 signal is a delayed clone of the Exsync (polarity, period, etc. are the same as for Exsync). |



**Parameterization of Time Values**

Operator *TrgBoxLine* is optimized for use with microEnable 5 products. If you work with microEnable IV frame grabbers, you have to double all parameter values that are messured in time units (e.g., µs).

| ExsyncPolarity | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter adjusts the polarity of the Exsync signal genarator to the polarity accepted by the connected camera. Use LowActive, if the camera opens the shutter on a falling edge, otherwise use HighActive. | |

| ImgTriggerMode | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | FreeRun |
| **Range** | {FreeRun, ExternSw_Trigger, ExternSw_Gate} |

This parameter selects the operation mode for the internal Image Gate.

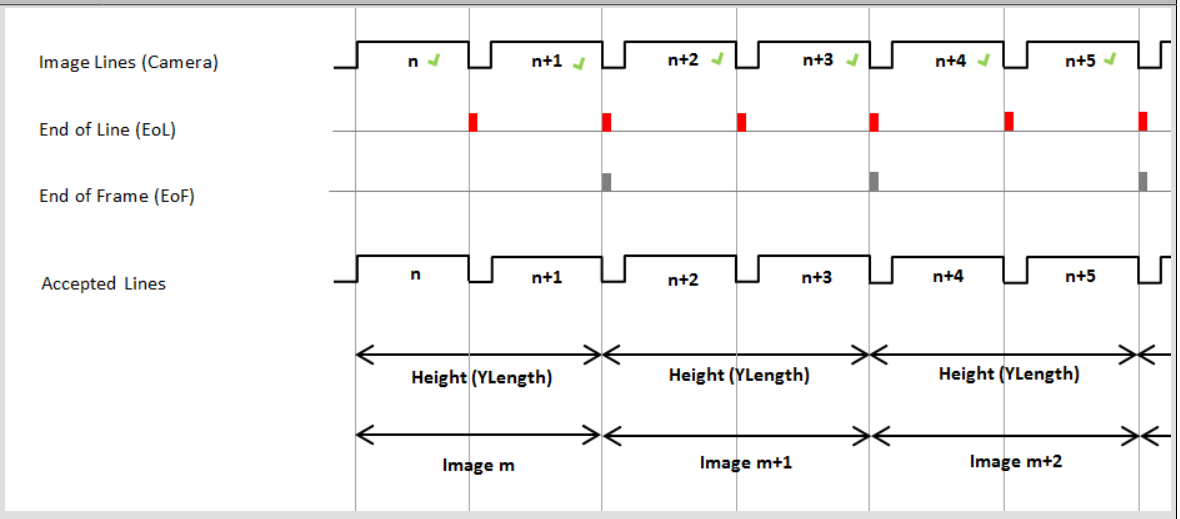
The image trigger input signal may be created by external (peripheral) devices (e.g., shaft encoder), or by software. The source for the external image trigger input you can select via the parameter *ImgTrgInSource*, see below).

The values of parameter ImgTriggerMode induce the following behaviour:

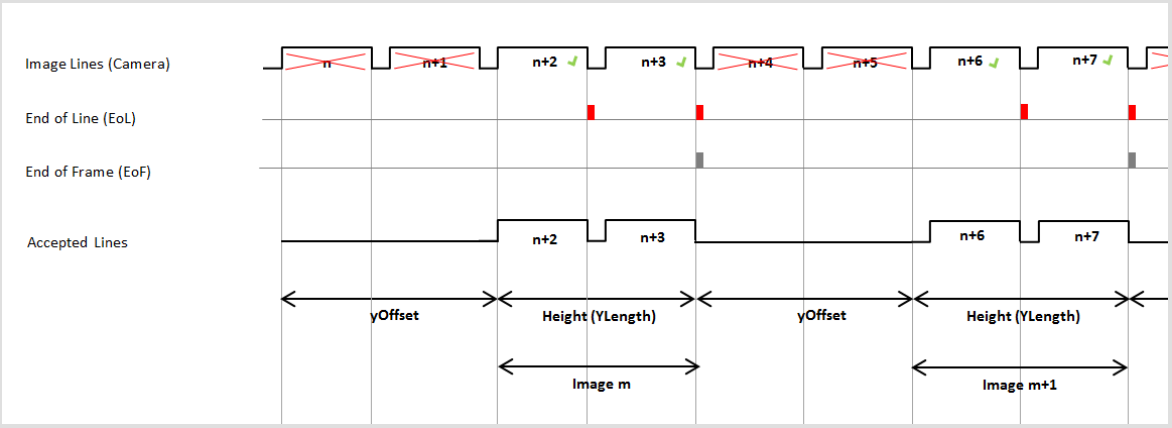
- **FreeRun**:
  - All incoming lines transmitted by the camera are accepted.
  - The Image Gate is ignored.
  - Parameter *YLength* defines when the end of frame (EOF) is generated.
  - If parameter *yOffset* is greater than Zero, a number of yOffset lines between two sequential images is ommitted.

    **Example**: *yOffset* = 0, *YLength* = 2:

**ImgTriggerMode**



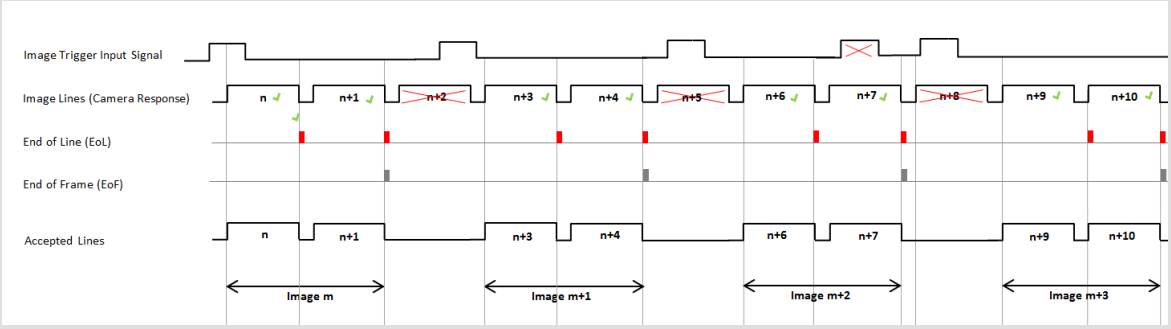

**Example**: *yOffset* = 2, *YLength* = 2:




- **ExternSw_Trigger**:
  - At the rising edge of an accepted image trigger input signal, a new frame is started and incoming lines are appended to an image up to *YLength*.
  - Parameter *YLength* defines when the end of frame (EOF) is generated.
  - A new rising edge of the image trigger is only accepted after *YLength* lines have been appended and EOF is generated. A new rising edge of the image trigger is ignored if it occurs before *YLength* lines have been appended to an image and EOF is generated.

    **Example**: *YLength* is set to value 2:

**ImgTriggerMode**

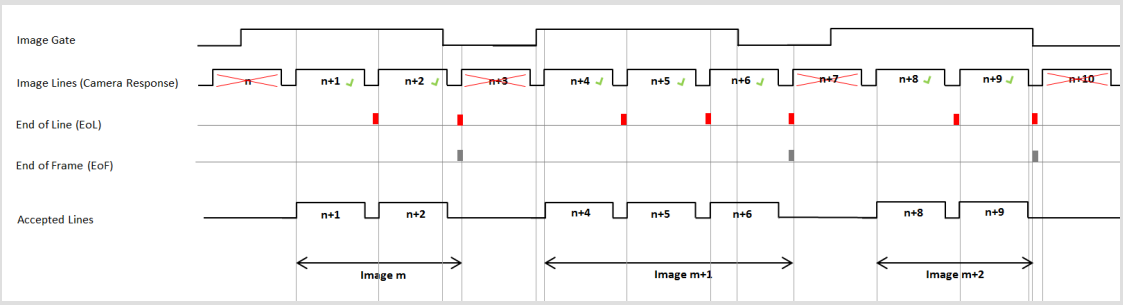



- **ExternSw_Gate**:
  - In gated mode, it is important how long the image trigger input signal is active, since it functions as the Image Gate signal.
  - The exact behaviour of this mode can be defined by the parameter *MaxGatedHeight*. (With *MaxGatedHeight* you can define if the maximum image height of the images is restricted or unrestricted. For details, see description of parameter *MaxGatedHeight*.)

    **Behaviour of mode ExternSW_Gate when parameter *MaxGatedHeight* is set to value "Unrestricted"**

    - While the Image Gate signal is active, the incoming lines are appended to an image.
    - The Image Gate controls which image lines are valid:
      - While the Image Gate signal is active, the incoming lines are accepted. (See figure, e.g., lines n+8 and n+9.)
      - While the Image Gate signal is not active, the incoming lines are ignored. (See figure, e.g., line n+3.)
      - Each line which starts while the Image Gate signal is active is valid. (See figure: n+1, n+2, n+4, n+5, n+6, n+8 and n+9 are valid.)

        If the falling edge of the Image Gate occurs before all the data of the last line are acquired, the line is nevertheless valid and acquired completely. (See figure: n+2 and n+6)
      - There are always complete lines being transferred.
      - Each line which starts while the Image Gate is down is not valid. (See figure: Line n and n +7 are not valid, line n+4 is valid.)

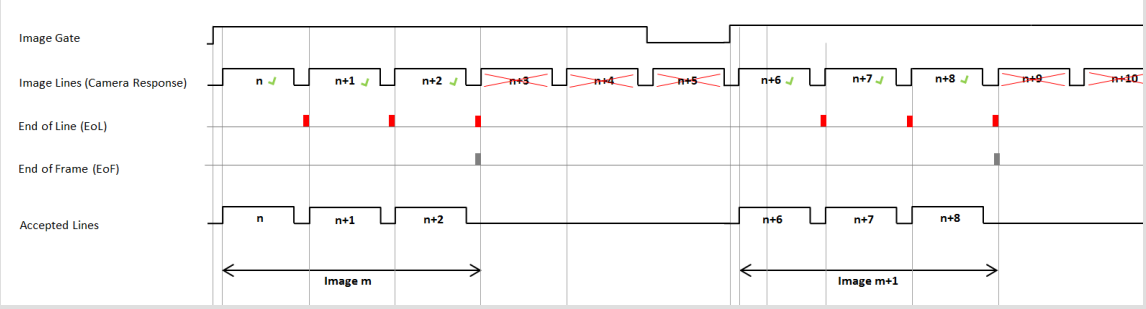



    - End of line (EoL) and end of frame (EoF) are either generated directly before the start of a new line, or at the falling edge of the Image Gate.

| ImgTriggerMode |
|---|
| *EoL directly before start of new line*: If the falling edge of a line occurs within the open Image Gate and the Image Gate covers the start of the next line, the EoL is generated directly before the rising edge of the next line. (See figure: n+1, n+4, n+5 and n+8.)<br><br>*EoL together with EoF directly before start of new line*: If the falling edge of the Image Gate occurs before the falling edge of a line, EoL and EoF are generated directly before the rising edge of the next line. (See figure: Image m and image m+1.)<br><br>*EoL together with EoF at falling edge of Image Gate*: If the falling edge of a line occurs within the open Image Gate and the falling edge of the Image Gate occurs before the rising edge of the next line, EoL and EoF are generated directly after the falling edge of the Image Gate. (See figure: Image m+2.) |

**Behaviour of Mode ExternSW_Gate when parameter *MaxGatedHeight* is set to value "Restricted"**
If parameter *MaxGatedHeight* is set to "restricted", the behaviour is the same as described above for *MaxGatedHeight* with value "unrestricted", with the following exceptions:

- As long as the image gate is open, the lines are appended up to a maximal image height of*YLength*. All following lines are ommitted until the next rising edge of the Image Gate signal.
- If a falling edge of the Image Gate signal occurs before YLength lines are appended: The falling edge of the Image Gate signal defines the height of the image, even if the image has not the height of YLength. (See parameter *MaxGatedHeight* for more information about the image height.)
- The example below (figure) shows the behaviour at *MaxGatedHeight* = restricted and *YLength* = 3.




| ImgTrgInSource | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | InSignal0 |
| **Range** | {InSignal0, InSignal1, InSignal2, InSignal3, InSignal4, InSignal5, InSignal6, InSignal7, SoftwareTrigger} |
| This parameter specifies the signal source which is used to trigger the image acquisition. This is only relevant if the ImgTriggerMode is set to ExternSw_Trigger or ExternSw_Gate. | |

| ImgTrgInPolarity | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter defines the polarity of the external input trigger signal. | |

| ImgTrgDelay | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 0 |
| **Range** | {0, 65535} |
| The parameter delays the image trigger signal by the given number of image lines. | |

| FlashEnable | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | OFF |
| **Range** | {OFF, ON} |
| Enables or disables the flash output. The pulse width of the flash signal is equal to one line period. | |

| FlashPolarity | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | LowActive |
| **Range** | {LowActive, HighActive} |
| The parameter defines the polarity for the generated Flash signal. | |

| FlashDelay | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 0 |
| **Range** | {0, 4095} |
| This parameter specifies the number of lines to delay the generated Flash signal, with respect to an external trigger input. Therefore, it is possible to synchronize the flash to the external trigger input.<br><br>The pulse width of the flash signal is equal to one line period. | |

| SoftwareTrgPulse | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | |
| **Range** | {1} |
| Setting this parameter to 1 will generate a software trigger. This is only relevant if the TriggerMode is set to an external trigger mode and ImgTrgInSource is set to SoftwareTrigger. | |

| SoftwareTrgInput | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | |
| **Range** | {0, 1} |
| With this parameter a software gate can be produced for the image trigger mode ExternSw_Gate. | |

| ImgTrgIsBusy | |
|---|---|
| **Type** | dynamic/static read parameter |
| **Default** | 0 |
| **Range** | {0, 1} |
| The ImgTrgIsBusy parameter enables software readout of the busy state for the image trigger. If busy then this parameter is set to 1 to reflect an ongoing image capture. If set to 0 then the operator is not busy. | |

| CC1output | |
|---|---|
| **Type** | dynamic/static write parameter |

| CC1output | |
|---|---|
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Exsync2, Exsync2Invert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC1 line of the CameraLink cable. | |

| CC2output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Exsync2, Exsync2Invert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC2 line of the CameraLink cable. | |

| CC3output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Exsync2, Exsync2Invert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC3 line of the CameraLink cable. | |

| CC4output | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | Exsync |
| **Range** | {Exsync, ExsyncInvert, Exsync2, Exsync2Invert, Flash, FlashInvert, Gnd, Vcc} |
| This parameter specifies the signal available at the CC4 line of the CameraLink cable. | |

| ImgTrgDebouncingMaxTime | |
|---|---|
| **Type** | static write parameter |
| **Default** | 65.520 us |
| **Range** | [0.016, 1000000] us |

This parameter specifies the maximal time for ImgTrgDebouncingTime parameter. The smaller the maximal time the less FPGA resources are required to implement the debouncing timer.



**Parameterization of Time Values**

Operator *TrgBoxLine* is optimized for use with microEnable 5 products. If you work with microEnable IV frame grabbers, you have to double all parameter values that are messured in time units (e.g., µs).

| ImgTrgDebouncingTime | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 0.112 us |
| **Range** | [0.016, ImgTrgDebouncingMaxTime] us |

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



**Parameterization of Time Values**

Operator *TrgBoxLine* is optimized for use with microEnable 5 products. If you work with microEnable IV frame grabbers, you have to double all parameter values that are messured in time units (e.g., µs).

| LineTrgDebouncingMaxTime | |
|---|---|
| **Type** | static write parameter |
| **Default** | 65.520 us |
| **Range** | [0.016, 1000000] us |

This parameter specifies the maximal time for LineTrgDebouncingTime parameter. The smaller the maximal time the less FPGA resources are required to implement the debouncing timer.

**Parameterization of Time Values**

Operator *TrgBoxLine* is optimized for use with microEnable 5 products. If you work with microEnable IV frame grabbers, you have to double all parameter values that are messured in time units (e.g., µs).

| LineTrgDebouncingTime | |
|---|---|
| **Type** | dynamic/static write parameter |
| **Default** | 0.112 us |
| **Range** | [0.016, ImgTrgDebouncingMaxTime] us |

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

**Parameterization of Time Values**

Operator *TrgBoxLine* is optimized for use with microEnable 5 products. If you work with microEnable IV frame grabbers, you have to double all parameter values that are messured in time units (e.g., µs).

### 30.5.4. Examples of Use

The use of operator TrgBoxLine is shown in the following examples:

- Section 12.15.8.2, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Operator **TrgBoxLine**'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design. For parameters of "TrgBoxLine" on microEnable4 IV VD4-CL/PoCL platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine'.

- Section 12.15.9.2, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Operator **TrgBoxLine**'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design. For parameters of "TrgBoxLine" on microEnable IV VQ4-GE/-GPoE platform please read corresponding operator documentation under Section 30.5, 'TrgBoxLine'.

- Section 12.15.10.2, ' Line Scan Trigger for microEnable 5 marathon/LightBridge VCL with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.11.2, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.12.2, 'Line Scan Trigger for microEnable 5 marathon VCX QP with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.13.2, 'Line Scan Trigger for imaFlex CXP-12 Quad with TrgBoxLine Operator Usage'

A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

- Section 12.15.14.2, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 with TrgBoxLine Operator Usage'

  A VisualApplets design example showing the usage of operator TrgBoxLine in a simple design.

# 30.6. Operator RGB2XYZ

**Operator Library: Prototype**



**Warning**

Disclaimer: This module is part of the prototype library. It is absolutely preliminary and subject to extensive changes, even removal. Therefore, this module must not be used for productive designs. It is included for testing purposes only.

The module RGB2XYZ converts the color space from an anticipated sRGB, white point D65 to CIE XYZ. The coefficients are parametrizable to allow customized transformations. Since the XYZ values range [0, 100] the output bit width is 7-bit, however if more bits are selected at the output link the values are scaled by 2^(n-7) to make a higher precision available.

The transformation matrix is defined by the parameters *TransMatrixX1* to *TransMatrixZ3*. These parameters are a 3x3 matrix containing double precision coefficients. The following formula represents the transformation algorithm.

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} TransMatrixX1 & TransMatrixX2 & TransMatrixX3 \\ TransMatrixY1 & TransMatrixY2 & TransMatrixY3 \\ TransMatrixZ1 & TransMatrixZ2 & TransMatrixZ3 \end{pmatrix} \times \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

## 30.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data output |

## 30.6.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 24 | [21, 63] |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | as I |
| **Color Flavor** | FL_RGB | FL_XYZ |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 30.6.3. Parameters

| TransMatrixX1 | |
|---|---|
| **Type** | static read/write parameter |
| **Default** | 0.4124 |
| **Range** | [0, 1] |

| TransMatrixX1 |
| --- |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. |

| TransMatrixX2 | |
| --- | --- |
| **Type** | static read/write parameter |
| **Default** | 0.3576 |
| **Range** | [0, 1] |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. | |

| TransMatrixX3 | |
| --- | --- |
| **Type** | static read/write parameter |
| **Default** | 0.1805 |
| **Range** | [0, 1] |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. | |

| TransMatrixY1 | |
| --- | --- |
| **Type** | static read/write parameter |
| **Default** | 0.2126 |
| **Range** | [0, 1] |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. | |

| TransMatrixY2 | |
| --- | --- |
| **Type** | static read/write parameter |
| **Default** | 0.7152 |
| **Range** | [0, 1] |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. | |

| TransMatrixY3 | |
| --- | --- |
| **Type** | static read/write parameter |
| **Default** | 0.0722 |
| **Range** | [0, 1] |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. | |

| TransMatrixZ1 | |
| --- | --- |
| **Type** | static read/write parameter |
| **Default** | 0.0193 |
| **Range** | [0, 1] |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. | |

| TransMatrixZ2 | |
| --- | --- |
| **Type** | static read/write parameter |
| **Default** | 0.1192 |
| **Range** | [0, 1] |

| TransMatrixZ2 |
| --- |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. |

| TransMatrixZ3 | |
| --- | --- |
| **Type** | static read/write parameter |
| **Default** | 0.9505 |
| **Range** | [0, 1] |
| Defines one coefficient of the transformation matrix RGB to XYZ. The value of this coefficient is a double value. | |

# 30.7. Operator XYZ2LAB

**Operator Library: Prototype**



**Warning**

Disclaimer: This module is part of the prototype library. It is absolutely preliminary and subject to extensive changes, even removal. Therefore, this module must not be used for productive designs. It is included for testing purposes only.

The module RGB2XYZ converts the color space from an anticipated CIE XYZ to L*A*B*. The coefficients are parametrizable to allow customized transformations.

## 30.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, image data input |
| **Output Link** | O, image data output |

## 30.7.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | {24, 30, 36} | as I |
| **Arithmetic** | signed | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I |
| **Color Format** | VAF_COLOR | as I |
| **Color Flavor** | FL_XYZ | FL_LAB |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 30.7.3. Parameters

None

# 31. Library Signal



The *signal* library includes operators for signal data processing such as trigger signals.

The following list summarizes all Operators of Library Signal

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | DelayToSignal | Delays the input signal. Delay is controlled by a input link. | Version 1.2 |
|  | Downscale | Reduces the input frequency by an adjustable factor. | Version 1.2 |
|  | EventToSignal | Generates a signal pulse for each input pixel with value 1. | Version 1.2 |
|  | FrameEndToSignal | Generates a signal pulse when the end of the input image is detected. | Version 1.2 |
|  | FrameStartToSignal | Generates a signal pulse when the start of an input image is detected. | Version 1.2 |
|  | Generate | Generates a periodic signal with controllable period time. | Version 1.2 |
|  | GetSignalStatus | Obtain the current value of a signal link. | Version 1.2 |
|  | Gnd | Provides a signal with the constant value 0 (LOW). | Version 1.2 |
|  | LimitSignalWidth | Limits the maximum pulse width of the input signal using a parameterizable maximum. | Version 1.2 |
|  | LineEndToSignal | Generates a signal pulse when the end of a input image line is detected. | Version 1.2 |
|  | LineStartToSignal | Generates a signal pulse when the start of an input image line is detected. | Version 1.2 |
|  | PeriodToSignal | Generates a periodic signal. Period time controlled by input link. | Version 1.2 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | PixelToSignal | Converts an image data stream into a signal stream. | Version 1.2 |
|  | Polarity | Controls the polarity of the signal (invert). | Version 1.2 |
|  | PulseCounter | Counts every occurrence of a one (high) at signal input link I. | Version 1.2 |
|  | RsFlipFlop | Implements a set-reset flip-flop. | Version 1.2 |
|  | RxSignalLink | Receives signals from a TxSignalLink operator in the design. | Version 2.0 |
|  | Select | Selects a signal source from N signal sources by parameter and forward selected signal to the output. | Version 1.2 |
|  | SetSignalStatus | Set a signal link status by use of a parameter. | Version 1.2 |
|  | ShaftEncoder | Analyzes shaft encoder signal traces and outputs encoder pulses as well as the direction. | Version 1.2 |
|  | ShaftEncoderCompensate | Compensates the rewind of a shaft encoder. | Version 1.2 |
|  | SignalDebounce | Suppresses fast changing signals at the input link with adjustable minimum time. | Version 1.2 |
|  | SignalDelay | Delays the input signal. Delay is controlled by a parameter. | Version 1.2 |
|  | SignalEdge | Generates a pulse of one design clock cycle, if a rising-, falling- or both- edges are detected at the input. | Version 1.2 |
|  | SignalGate | Gates the image stream between I and O by use of a signal input. | Version 1.2 |
|  | SignalToDelay | Measures and outputs the delay between two signals. | Version 1.2 |
|  | SignalToPeriod | Measures and outputs the period time of the input signal. | Version 1.2 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | SignalToPixel | Converts the input signal stream into a 0D pixel stream. | Version 1.2 |
|  | SignalToWidth | Measures and outputs the pulse width of the input signal. | Version 1.2 |
|  | SignalWidth | Generates an output pulse with controllable with for rising edges at the input. | Version 1.2 |
|  | SyncSignal | Synchronizes a number of input links to a master signal. | Version 1.2 |
|  | TxSignalLink | Sends signals to any RxSignalLink operator in the design. | Version 2.0 |
|  | Vcc | Provides a signal with the constant value 1 (HIGH). | Version 1.2 |
|  | WidthToSignal | Defines the width of a pulse. Width is controlled by a input link. | Version 1.2 |

Table 31.1. Operators of Library Signal

# 31.1. Operator DelayToSignal

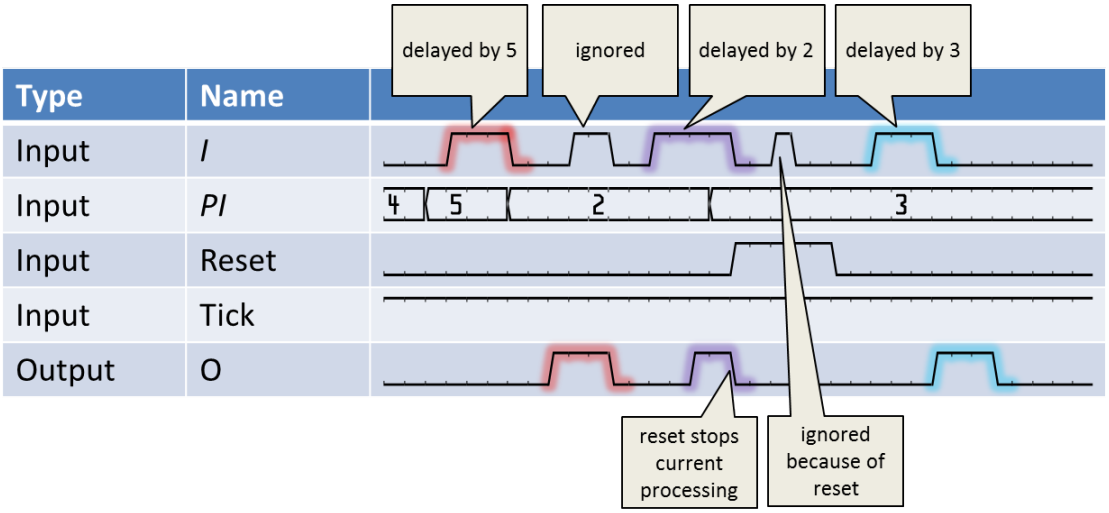
**Operator Library: Signal**

The operator delays the input signal and provides it on its output. The delay is controlled by the control data input link PI.

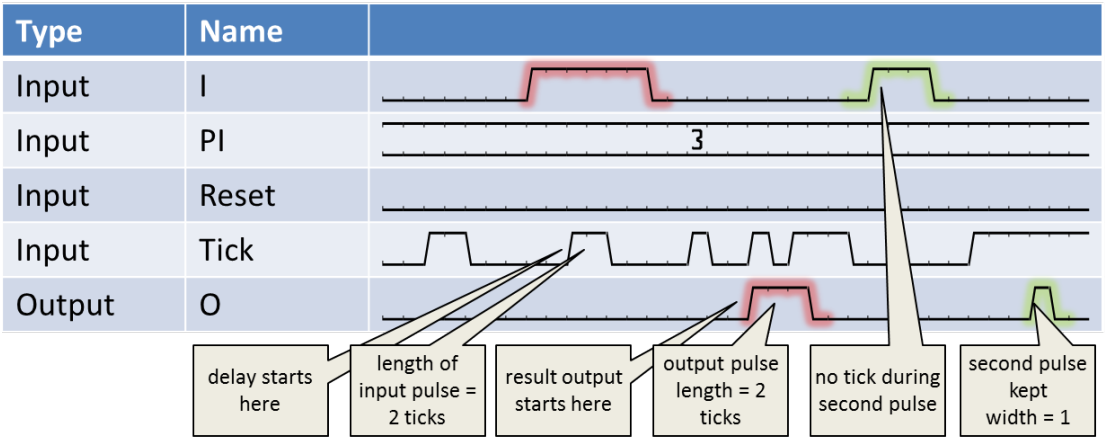
The rising edge of a pulse at the input starts the delay. The delay time is equal to the last valid value at input link PI and is measured in ticks being high. The pulse width of the input pulse is kept, i.e. the rising and falling edges are delayed.

During delaying of a pulse no new pulses at the operator input can be accepted. In this case, every new input pulse will be ignored. Moreover, a change of the value at PI does not change the current pulse processing. PI is sampled at the start of the delay.

The operator can be reseted using input link *Reset*. While the reset input is high, no output pulses are processed. Any processing is aborted. The operator restarts operation when the reset input is low. The following waveform illustrates the operator's behavior.




The Tick input defines the time, the operator is processing data. It can be used like a prescaler. In most cases, the Tick input is not required. Tie it to operator VCC in this case. In the following figure, the influence of the Tick input is shown.




One special case when using ticks is that input pulses are sampled even if no tick is present. This is shown for the second input pulse of the waveform. This ensures that no input pulses can get lost.

This operator is excluded from the VisualApplets functional simulation.

### 31.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, signal data input<br>PI, control image data input<br>Tick, signal data input<br>Reset, signal data input |
| **Output Link** | O, signal data output |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.
- Input link *PI* is asynchronous to the signal inputs.

### 31.1.2. Supported Link Format

| Link Parameter | Input Link I | Input Link PI | Input Link Tick |
|---|---|---|---|
| **Bit Width** | 1 | [1, 64] | 1 |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any | any |
| **Max. Img Height** | any | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 31.1.3. Parameters

None

### 31.1.4. Examples of Use

The use of operator DelayToSignal is shown in the following examples:

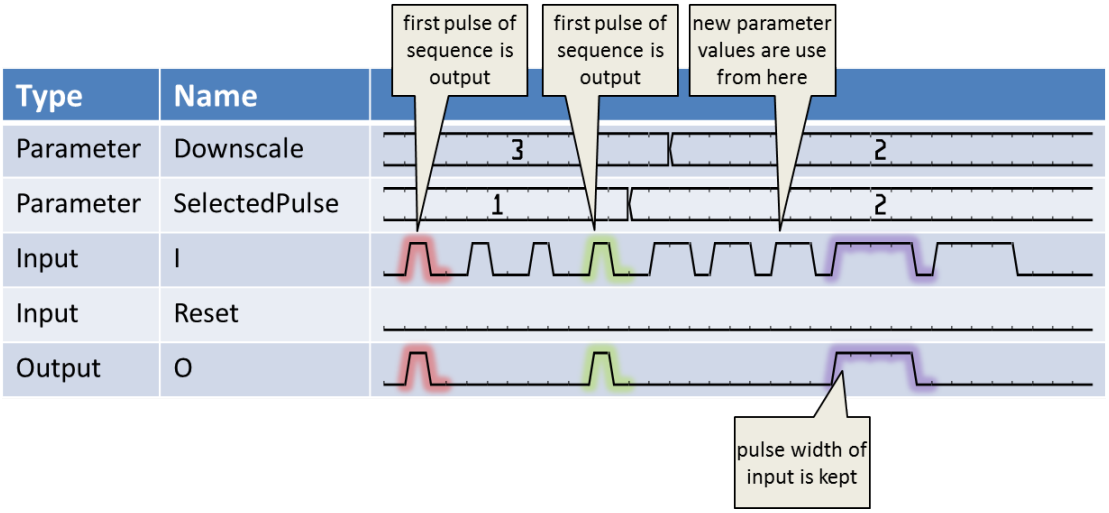
- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

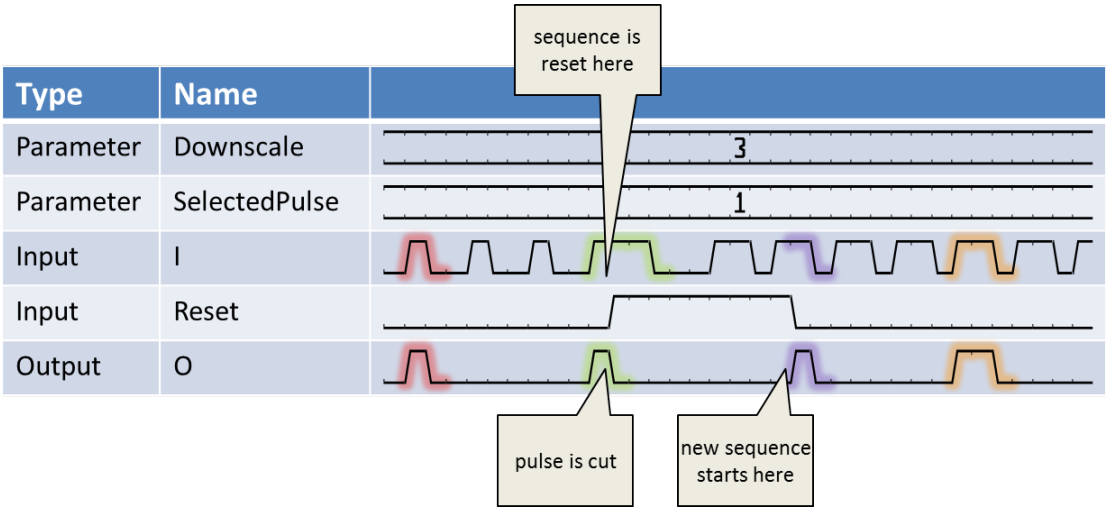
## 31.2. Operator Downscale

**Operator Library: Signal**

The operator downscales the input signal to the specified frequency, i.e. only every Nth pulse will be forwarded to the output. Parameter *Downscale* specifies the downscale factor. Parameter *SelectedPulse* defines which pulse of a downscale sequence is used and forwarded to the output. Parameter value changes are applied only when the operator starts a new sequence. In the following waveform, the operator behavior is visualized.




The operator can be reseted using input link Reset. While the reset input is high, no pulses are processed. Any processing is aborted. The operator restarts its operation when the reset input is low. The following waveform illustrates the operator's behavior.




This operator is excluded from the VisualApplets functional simulation.

### 31.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, signal data input<br>Reset, signal data input |
| **Output Link** | O, signal data output |

### 31.2.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Reset | Output Link O |
|---|---|---|---|
| **Bit Width** | 1 | 1 | as I |
| **Arithmetic** | unsigned | unsigned | as I |
| **Parallelism** | 1 | 1 | as I |
| **Kernel Columns** | 1 | 1 | as I |
| **Kernel Rows** | 1 | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | FL_NONE | as I |
| **Max. Img Width** | any | any | as I |
| **Max. Img Height** | any | any | as I |

### 31.2.3. Parameters

| **MaxScaleBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 8 |
| **Range** | [1, 32] |
| Defines the maximum range of parameter *Downscale*. This parameter is enabled only if *Downscale* is set to dynamic. | |

| **Downscale** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | [1, 2^MaxScaleBits-1] if dynamic, [1, 2^64-1] else |
| The actual downscale factor is defined with using this parameter. A downscale factor of one will forward any input pulse do the output.<br><br>The value has to be >= parameter *SelectedPulse*. | |

| **SelectedPulse** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | [1, 2^MaxScaleBits-1] if dynamic, [1, 2^64-1] else |
| This parameter selects the pulse of a sequence to be forward to the output.<br><br>The value has to be <= parameter *Downscale*. | |

### 31.2.4. Examples of Use

The use of operator Downscale is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

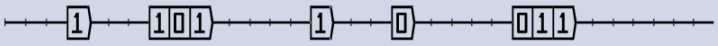
- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

Examples - Demonstration of how to use the operator

# 31.3. Operator EventToSignal

**Operator Library: Signal**

The operator generates a signal pulse for each input pixel with value 1. The operator input is a binary (1 bit) image stream. If a pixel with value one is present at the input of the operator, the operator will generate a signal pulse of one clock cycle. Keep in mind that image data streams include gaps between pixels. During a gap period, the operator does not generate output pulses. If you rather like to convert the last input binary pixel value into a signal, use operator *PixelToSignal* instead.

| Type | Name | |
|---|---|---|
| Input | I |  |
| Output | O |  |

This operator is excluded from the VisualApplets functional simulation.

## 31.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, signal data output |

## 31.3.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

## 31.3.3. Parameters

None

## 31.3.4. Examples of Use

The use of operator EventToSignal is shown in the following examples:

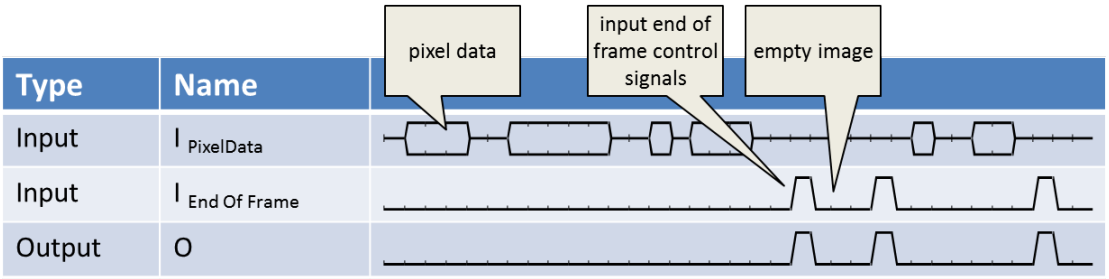
- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 31.4. Operator FrameEndToSignal

**Operator Library: Signal**

The operator generates a signal pulse when the end of the input image is detected. The operator input is an image stream. The operator waits for the end of the input image stream. When the frame end is detected, a signal pulse of one clock cycle is output. Note that the end of a frame might not occur directly after the last frame pixel. See the following figure for explanation:




Note that empty images can exist in VisualApplets. The operator will generate a signal pulse for empty frames, too.

This operator is excluded from the VisualApplets functional simulation.

## 31.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, signal output |

## 31.4.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | 1 |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | VALT_IMAGE2D | VALT_SIGNAL |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

## 31.4.3. Parameters

None

### 31.4.4. Examples of Use

The use of operator FrameEndToSignal is shown in the following examples:

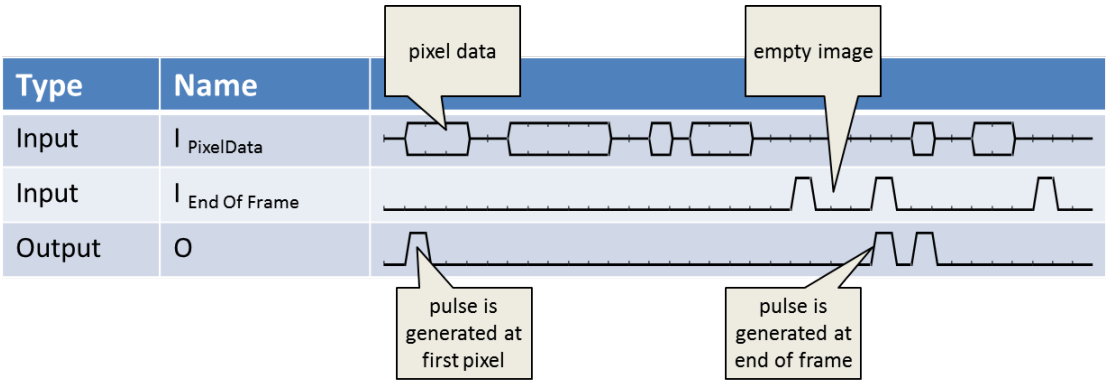
- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

# 31.5. Operator FrameStartToSignal

**Operator Library: Signal**

The operator generates a signal pulse when the start of an input image is detected. The operator input is an image stream. The operator waits for the first pixel of the input image. When the first pixel is detected, a signal pulse of one clock cycle is output. See the following figure for explanation:




Note that empty images can exist in VisualApplets. For empty images, the operator cannot find a first pixel. In this case, the operator will output a pulse at the end of the frame. If the first line of a frame does not include any pixels, the operator will output a pulse at the end of the first line.

This operator is excluded from the VisualApplets functional simulation.

## 31.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, signal output |

## 31.5.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | 1 |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | VALT_IMAGE2D | VALT_SIGNAL |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

## 31.5.3. Parameters

None

### 31.5.4. Examples of Use

The use of operator FrameStartToSignal is shown in the following examples:

- Section 13.3, 'Functional Example for Specific Operators of Library Memory and Library Signal'

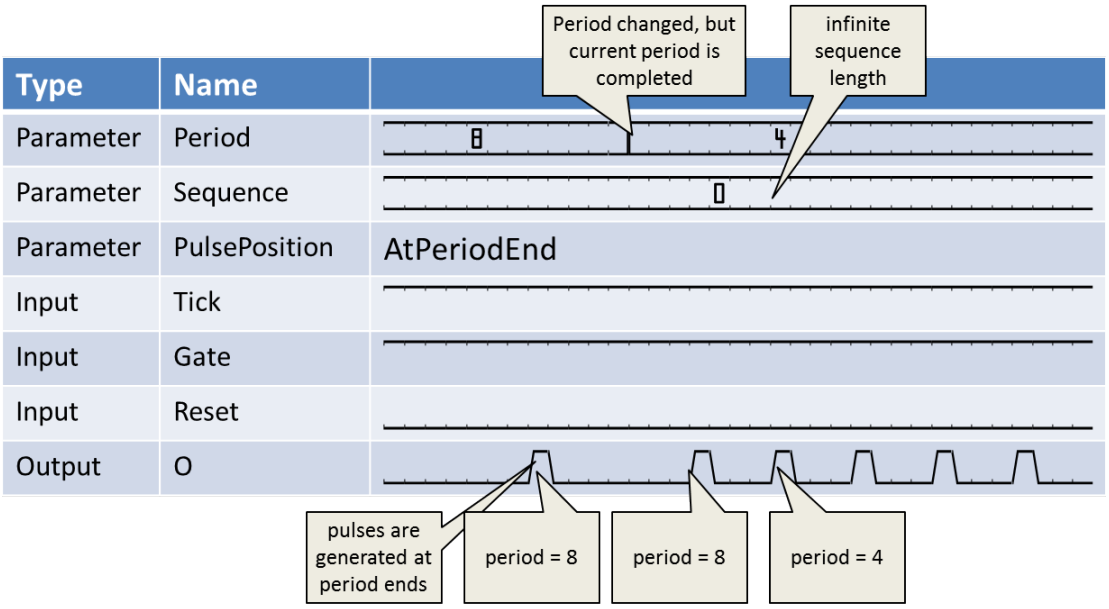
  Examples - Demonstration of how to use the operator

## 31.6. Operator Generate

**Operator Library: Signal**

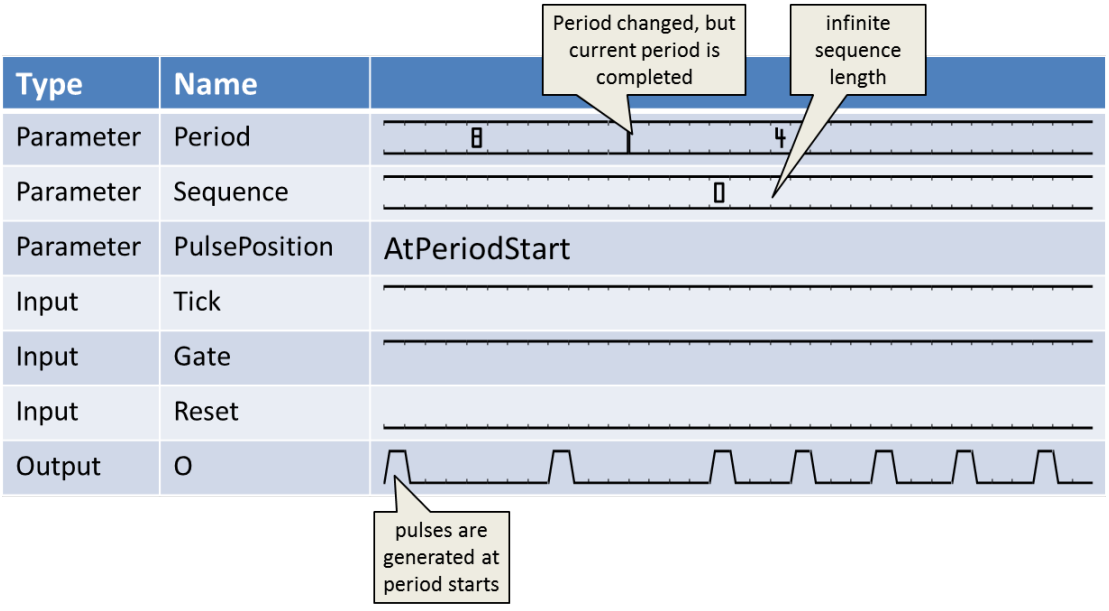
The operator generates a periodic signal at its output. Moreover, the pulse generation can be controlled by input signals. The period time and the number of pulses to be generated can be defined using parameters.

The period is defined using the parameter *Period*. The period is dynamic. Its range is defined by the parameter *PeriodBits*. In the following figure, the generation of pulses with period lengths eight and four are shown.

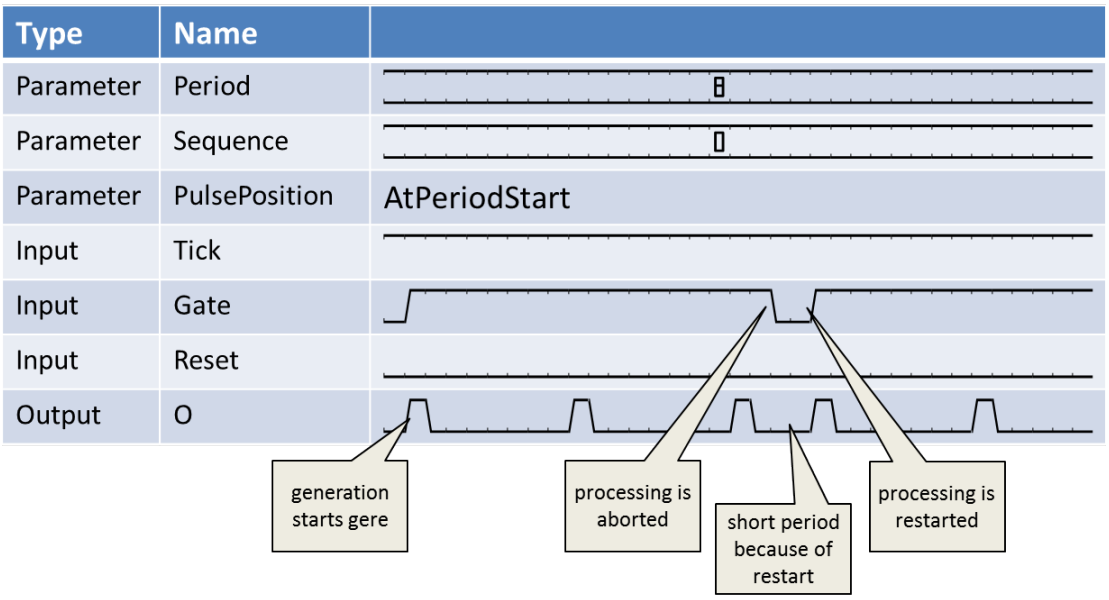



Changes at the *Period* parameter will only be applied to the operator when the generation of the current period is completed. This ensures periods in the parametrized lengths only.

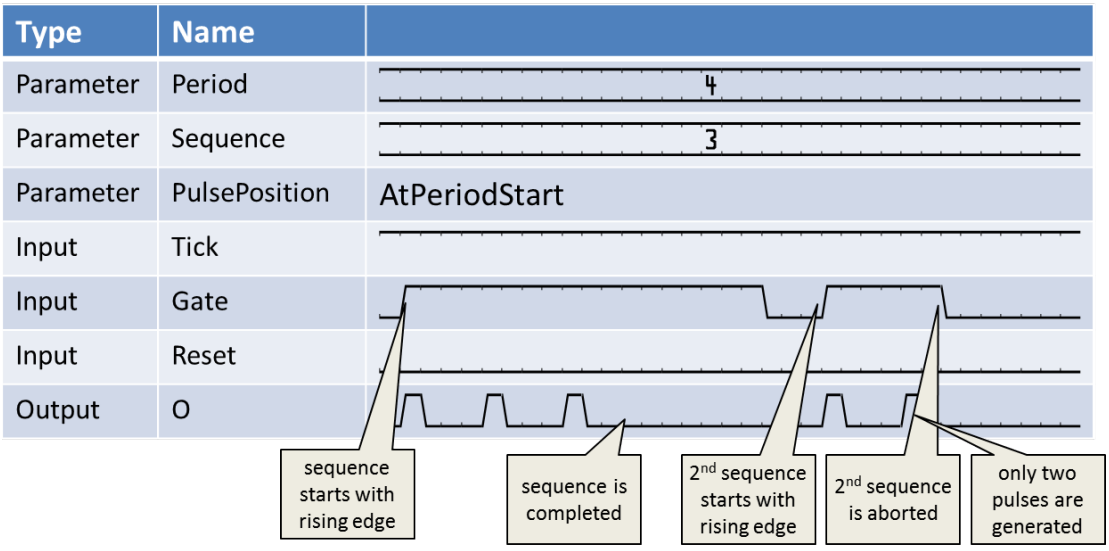
The pulse generation can either be at the end of the period or at the start as is controlled using the parameter *PulsePosition*. The next figure shows the generation at the start of a sequence.



By use of the *Gate* input link, the generation can be controlled. Periods are generated only if input *Gate* is active (high). The rising edge at the *Gate* input starts the period generation. A falling edge at the *Gate* input link immediately aborts the generation. Currently processed periods might not be completed in this case. Moreover, a new rising edge can yield a short period. This is shown in the following waveform.




The parameter *SequenceLength* facilitates the generation of sequences. In the previous figures, the parameter was set to 0 which results in an infinite sequence length. If you set the parameter to values > 0, the operator will only generate the number of parametrized periods. In the following waveform, the sequence length is set to 3 with a period set to 4. As can be seen in the waveform, the period generation is stopped after the third period. A new rising edge at the *Gate* input will start a new sequence. Sequences can be aborted if *Gate* turns to 0.




In many applications, an infinite sequence generation of the operator is required. In this case, set parameter *SequenceLength* to 0 and connect operator VCC to the *Gate* input. The operator will then start its period generation at process start e.g. acquisition start.

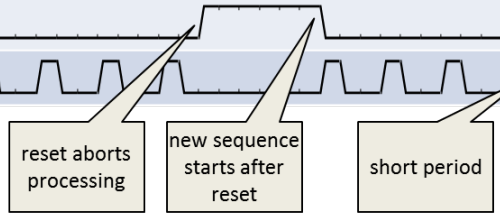
The period time is measured in *Ticks* being high. *Tick* is a signal input and can be used like a pre-scaler. For every high value at the *Tick* input, the period time is counted. The following waveform shows the

behavior of the *Tick* input to the period generation. In most cases, the *Tick* input is not required. Tie it to operator VCC in this case.

| Type | Name | |
|---|---|---|
| Parameter | Period | 2 4 |
| Parameter | Sequence | 0 |
| Parameter | PulsePosition | AtPeriodStart |
| Input | Tick | |
| Input | Gate | |
| Input | Reset | |
| Output | O | |

The *Reset* input link is used to reset the operator. This is useful if a long period or sequence has to be cancelled. In the next figure, the use of *Reset* is illustrated.

| Type | Name | |
|---|---|---|
| Parameter | Period | 3 |
| Parameter | Sequence | 0 |
| Parameter | PulsePosition | AtPeriodStart |
| Input | Tick | |
| Input | Gate | |
| Input | Reset | |
| Output | O | |






**Use the *Reset* input instead of setting *Period* to 0**

If *Period* is set to 0, a change of the period back to an active state may take very long (2^PeriodBit-1 clock cycles). It is therefore recommended to use the *Reset* input in this cases.

This operator is excluded from the VisualApplets functional simulation.

### 31.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | Tick, signal input<br>Gate, signal input<br>Reset, signal input |

| Property | Value |
|---|---|
| **Output Link** | O, signal input |

### 31.6.2. Supported Link Format

| Link Parameter | Input Link Tick | Input Link Gate |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | as Gate |
| **Max. Img Height** | any | as Gate |

### 31.6.3. Parameters

| **PeriodBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1, 64] |
| Number of bits required to encode the value specified by *Period*. This parameter is enabled only if parameter *Period* is dynamic. | |

| **Period** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 65535 |
| **Range** | [0, 2^PeriodBits-1] if dynamic, [1,2^64-1] if static |
| Defines the period time of the generated pulses. Do not use the following cases: If period set to 1, the operator will output a constant one without any gaps. Number of bits required to encode the value specified by If set to 0, the operator will not output any pulses. | |

| **SequenceBits** | |
|---|---|
| **Type** | static parameter |

| SequenceBits | |
|---|---|
| **Default** | 16 |
| **Range** | [1, 64] |
| Number of bits required to encode the value specified by *Sequence*. This parameter is enabled only if parameter *Sequence* is dynamic. | |

| Sequence | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [1, 2^SequenceBits-1] if dynamic, [1,2^64-1] if static |
| Defines the sequence length i.e. the number of pulses generated for each pulse at the *Gate* input link. | |

| PulsePosition | |
|---|---|
| **Type** | static parameter |
| **Default** | AtPeriodEnd |
| **Range** | {AtPeriodStart, AtPeriodEnd} |
| The output pulse can either be at the start of each period or at the end of each period. | |

### 31.6.4. Examples of Use

The use of operator Generate is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 31.7. Operator GetSignalStatus

**Operator Library: Signal**

Operator GetSignalStatus allows the monitoring of up to 64 input signal links by a software application. The current values at the link are stored and can be read using parameter *Status* from software. The value is bit-coded. Every bit represents one input link. Bit 0 input link 0, bit 1 input link 1, etc.

Different modes to read the link values are supported. The mode is selected using parameter *Mode*.

- In **`Direct`** mode the signal status is read out directly without and storing of previous values.

  In **`Rise`** mode, the status parameter is set to high if previously a rising edge has been detected at the input link.

  In **`Fall`** mode, the status parameter is set to high if previously a falling edge has been detected at the input link.

  In **`Edge`** mode, the status parameter is set to high if previously a rising or a falling edge has been detected at the input link.

  In **`Pulse`** mode, the status parameter is set to high if previously a rising AND a falling edge have been detected at the input link.

Reading the parameter causes a reset of the parameter value to zero.

This operator is excluded from the VisualApplets functional simulation.

## 31.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I0...I64, image data input |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.

## 31.7.2. Supported Link Format

| Link Parameter | Input Link I0...I64 |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 31.7.3. Parameters

| Mode | |
|---|---|
| **Type** | static parameter |
| **Default** | Direct |
| **Range** | {Direct, Rise, Fall, Edge, Pulse} |
| This parameter specifies the monitoring type mode. See explanations above. | |

| Status | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 2^NoOfInputLinks] |
| Shows the current link status for all inputs. The status is reset to 0 while reading. Bit 0 corresponds to input 0, bit 1 to input 1, etc. | |

### 31.7.4. Examples of Use

The use of operator GetSignalStatus is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

  Examples - Demonstration of how to use the operator

# 31.8. Operator Gnd

**Operator Library: Signal**

This operator provides a signal with constant value 0 (LOW).

This operator is excluded from the VisualApplets functional simulation.

## 31.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, signal output |

## 31.8.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 31.8.3. Parameters

None

## 31.8.4. Examples of Use

The use of operator Gnd is shown in the following examples:

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

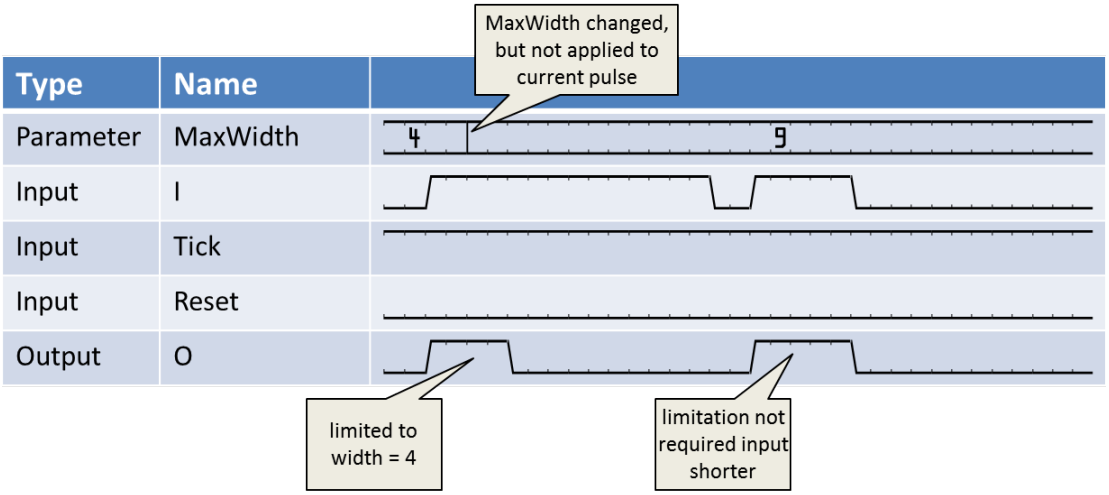
- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

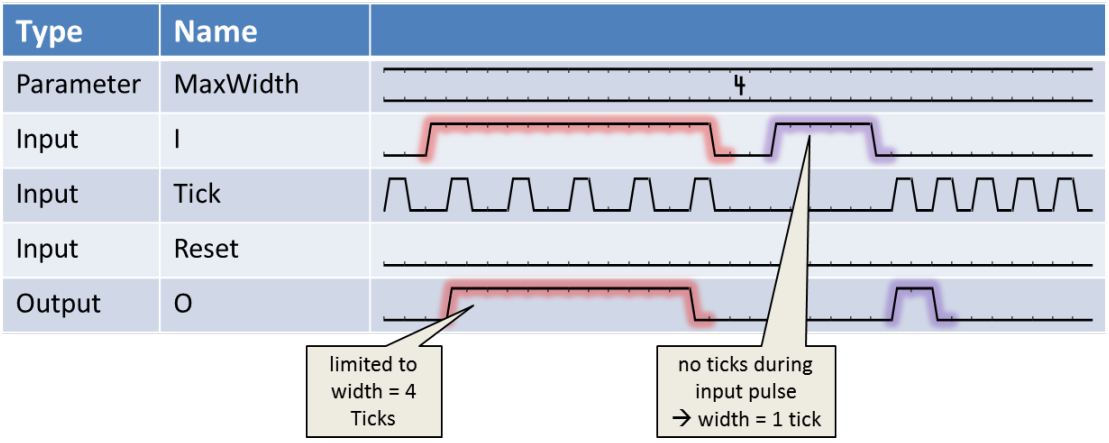
# 31.9. Operator LimitSignalWidth

**Operator Library: Signal**

The operator limits the pulse width of the input signal. To set the maximum width use parameter *MaxWidth*. The pulse width is the time, the pulse is 1 (HIGH). The following waveform shows the behavior of the operator. The first pulse if limited by the operator to four clock cycles defined by parameter *MaxWidth*. The second pulse is shorter than *MaxWidth* and therefore, no limitation is required. Parameter changes are applied to the next pulse and will not change the limitation of a currently processed pulse.




The pulse width is measured in Ticks. Tick is an input link. For every 1 (HIGH) at the Tick input the operator counts the width. This behavior is shown in the next figure. In most applications, the Tick input is not required. Tie it to Vcc in these cases.




An additional Reset input enables the reset of the operator. Any current pulse processing is aborted.

This operator is excluded from the VisualApplets functional simulation.

## 31.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, signal input<br>Tick, signal input<br>Reset, signal input |

| Property | Value |
|---|---|
| **Output Link** | O, signal output |

### 31.9.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 31.9.3. Parameters

| **MaxWidthBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1, 64] |
| The maximum possible value of *MaxWidth* is defined using this parameter. This parameter is enabled only if *MaxWidth* is set to dynamic. | |

| **MaxWidth** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 32768 |
| **Range** | [0,2^MaxWidthBits-1] if dynamic, [0, 2^64-1] if static |
| Defines the maximum signal width. | |

### 31.9.4. Examples of Use

The use of operator LimitSignalWidth is shown in the following examples:

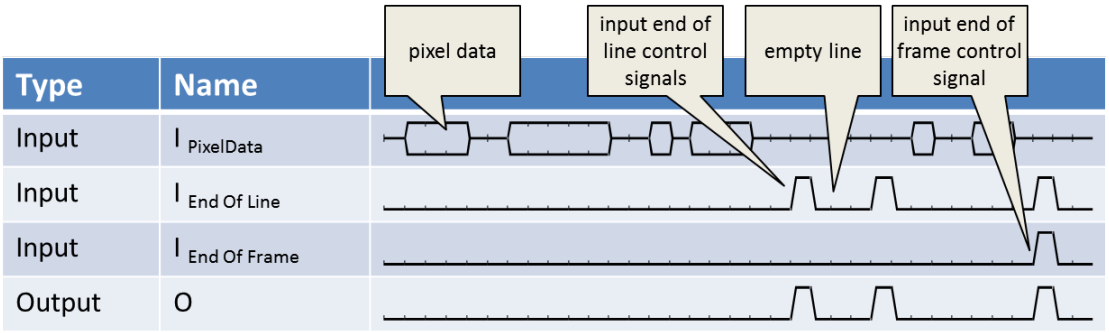
- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

  Examples - Demonstration of how to use the operator

# 31.10. Operator LineEndToSignal

**Operator Library: Signal**

The operator generates a signal pulse when the end of a input image line is detected. The operator input is an image stream. The operator waits for the end of lines at the input image stream. When the line end is detected, a signal pulse of one clock cycle is output. Note that the end of a line might not occur directly after the last line pixel. See the following figure for explanation:




The end of a frame always occurs at the same time as the end of the last frame line. Note that empty lines can exist in VisualApplets. The operator will generate a signal pulse for empty lines, too.

This operator is excluded from the VisualApplets functional simulation.

## 31.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, signal output |

## 31.10.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | 1 |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | VALT_SIGNAL |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

## 31.10.3. Parameters

None

### 31.10.4. Examples of Use

The use of operator LineEndToSignal is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

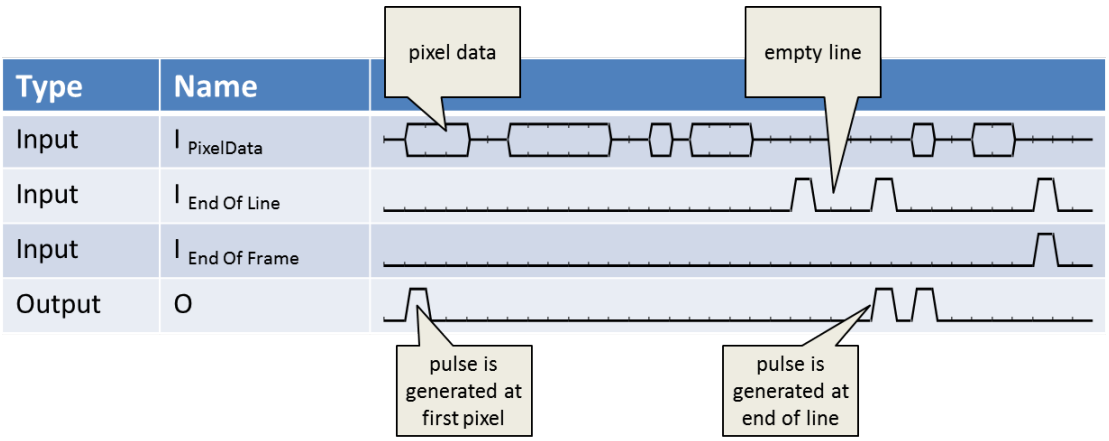
- Section 13.4, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

# 31.11. Operator LineStartToSignal

**Operator Library: Signal**

The operator generates a signal pulse when the start of an input image line is detected. The operator input is an image stream. The operator waits for the first pixel at each input image line. When the first pixel is detected, a signal pulse of one clock cycle is output. See the following figure for explanation:




Note that empty line can exist in VisualApplets. For empty lines, the operator cannot find a first pixel. In this case, the operator will output a pulse at the end of the line.

This operator is excluded from the VisualApplets functional simulation.

## 31.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, signal output |

## 31.11.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | 1 |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | VALT_SIGNAL |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

### 31.11.3. Parameters

None

### 31.11.4. Examples of Use

The use of operator LineStartToSignal is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 13.4, 'Functional Example for Specific Operators of Library Memory and Library Signal'

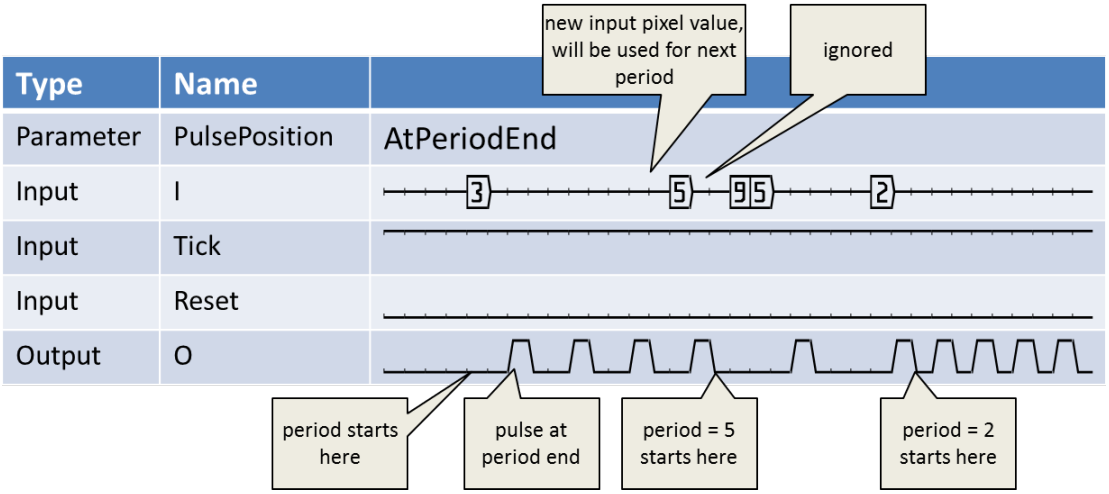
  Examples - Demonstration of how to use the operator

# 31.12. Operator PeriodToSignal

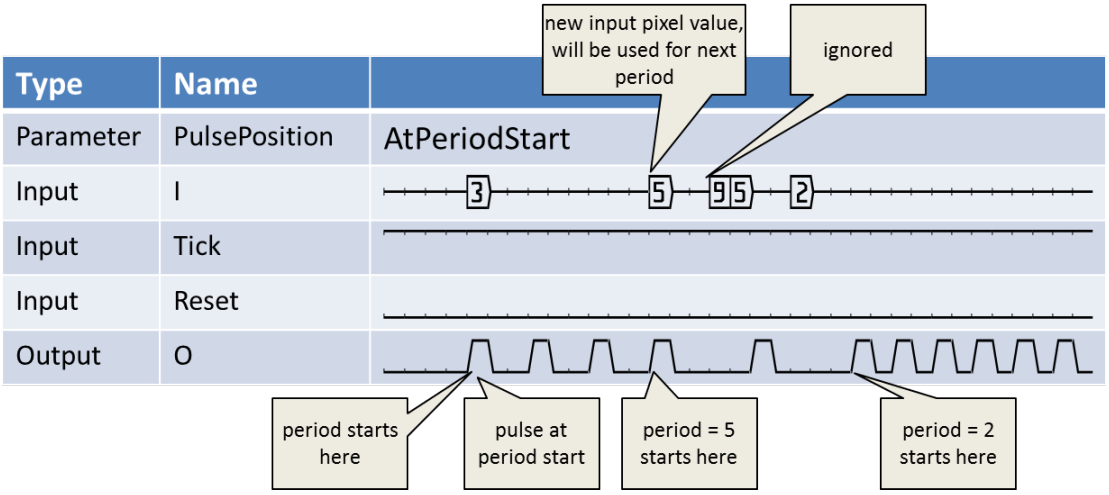
**Operator Library: Signal**

The operator generates a periodic signal at its output. The period time is controlled by the pixel value at the input link I. Thus the operator converts the input stream values into a signal with the period provided by the pixel values. The high time of the output pulses is one tick.

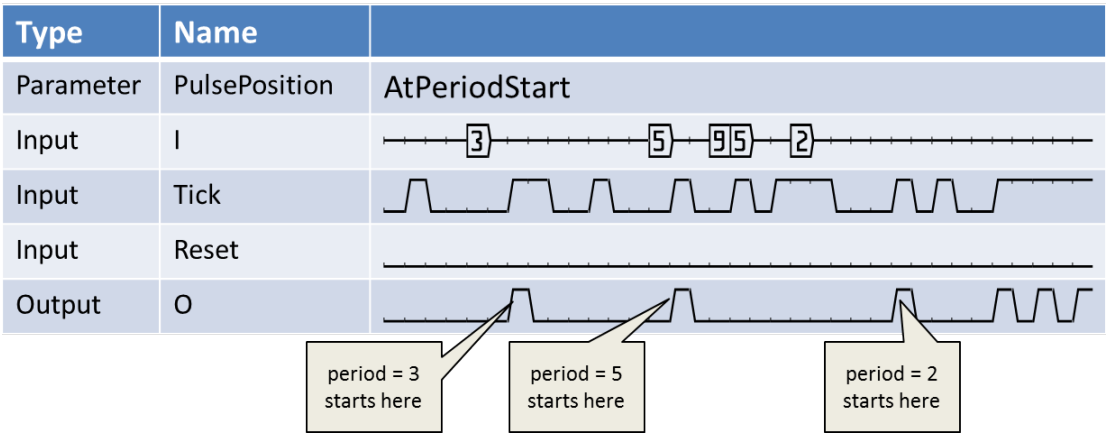
During processing of a current signal generation with a specified period, the operator will ignore any new values at the input. After the processing of a period is finished, the generates a new period with the length of the last valid input pixel value. The pulse position can either be at the beginning or at the end of a period and is controlled using parameter *PulsePosition*. In the following figure, the operator's behavior is illustrated.



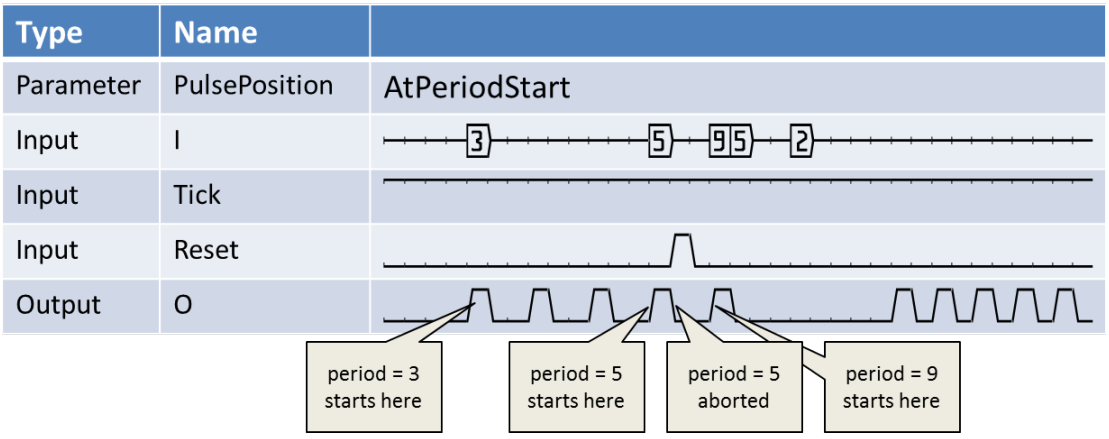

The next figure shows the generation of the pulse at the start of a period.




The period time is measured in Ticks being high. Tick is a signal input and can be used like a prescaler. For every high value at the Tick input, the period time is counted. The following waveform shows the behavior of the Tick input to the period generation. In most cases, the Tick input is not required. Tie it to operator VCC in this case.




The Reset input link is used to reset the operator. This is useful if a long period or sequence has to be cancelled. In the next figure, the reset usage is illustrated.




This operator is excluded from the VisualApplets functional simulation.

## 31.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, pixel data input<br>Tick, signal input<br>Reset, signal input |
| **Output Link** | O, signal input |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.
- Input link *I* is asynchronous to the signal inputs.

## 31.12.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Bit Width** | [1, 64] | 1 |

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 31.12.3. Parameters

| PulsePosition | |
|---|---|
| **Type** | static parameter |
| **Default** | AtPeriodEnd |
| **Range** | {AtPeriodStart, AtPeriodEnd} |
| The output pulse can either be at the start of each period or at the end of each period. | |

### 31.12.4. Examples of Use

The use of operator PeriodToSignal is shown in the following examples:

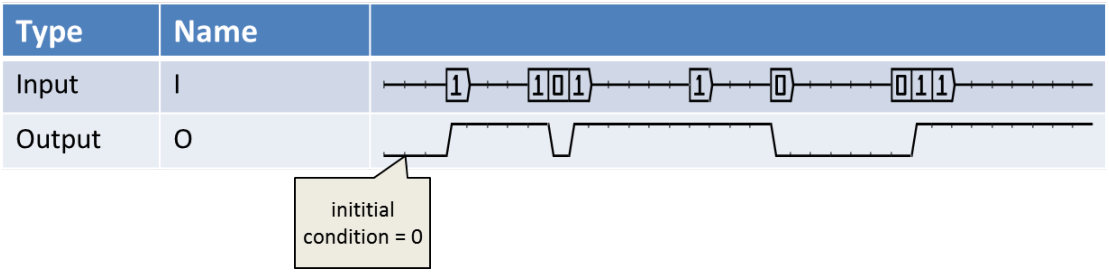
- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 31.13. Operator PixelToSignal

**Operator Library: Signal**

The operator converts an image data stream at the input into a signal stream. The input is a binary (1 bit) image stream. For every pixel with value zero at the input a low signal (zero) will be output. For every pixel with value one at the input, a high signal (one) will be output. Image data streams include gaps between pixel. The operator fills these gaps at the output with the last valid input pixel value. See the following figure for explanations.




This operator is excluded from the VisualApplets functional simulation.

## 31.13.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Link** | O, signal data output |

## 31.13.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

## 31.13.3. Parameters

None

## 31.13.4. Examples of Use

The use of operator PixelToSignal is shown in the following examples:

- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 31.14. Operator Polarity

**Operator Library: Signal**

The operator Polarity controls the output link polarity. Depending on the parameter setting the output signal can either be bypassed or inverted. See the following illustration.

| Type | Name | |
|---|---|---|
| Parameter | Invert | NotInvert \| Invert |
| Input | I |  |
| Output | O |  |

This operator is excluded from the VisualApplets functional simulation.

## 31.14.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, signal data input |
| **Output Link** | O, signal data output |

## 31.14.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 31.14.3. Parameters

| **Invert** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | NotInvert |
| **Range** | {NotInvert, Invert} |
| This parameter specifies the polarity of the output signal. NotInvert = the input is bypassed to the output. Invert = the output signal is an inverted copy of the input signal. | |

## 31.14.4. Examples of Use

The use of operator Polarity is shown in the following examples:

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

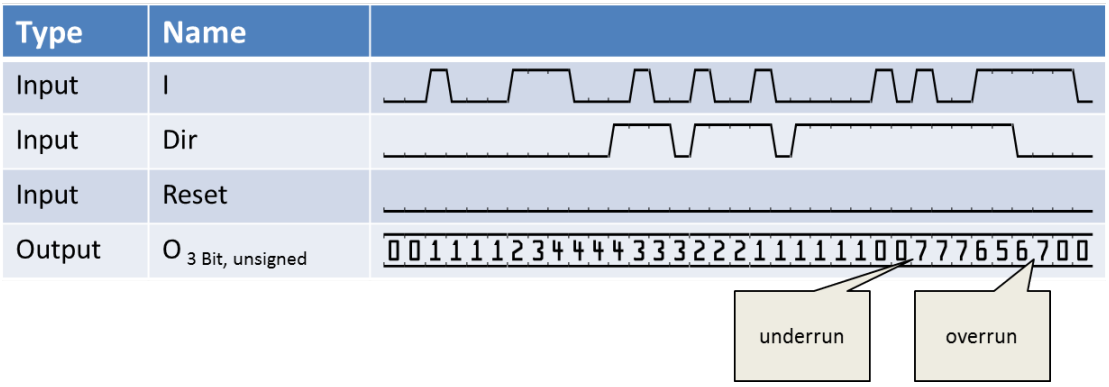
  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 31.15. Operator PulseCounter

**Operator Library: Signal**

This operator is an up/down counter. For every occurrence of a one (high) at the signal input link I, the operator is incremented or decremented. It is incremented (count up) if input link Dir = 0 and decremented (count down) if input link Dir = 1. The counter can be reset using input link Reset.

The operator output is a VALT_PIXEL0D pixel stream. At each clock cycle a pixel is output representing the current counter value. In the following figure, the behavior of the operator is illustrated.




The waveform shows underrun and overrun situations. In these cases, the operator is further decreased or increased even if the minimum or maximum of the counter has been reached. In this case, the counter will jump to the maximum or minimum possible value representable at the output.

The additional reset input link allows the reset of the operator. All outputs are set to zero, while reset.

This operator is excluded from the VisualApplets functional simulation.

## 31.15.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, signal input<br>Dir, signal input<br>Reset, signal input |
| **Output Link** | O, counter value pixel data output |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.

## 31.15.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Dir |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |

| Link Parameter | Input Link I | Input Link Dir |
|---|---|---|
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | [1, 64] |
| **Arithmetic** | unsigned | {unsigned, signed} |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_PIXEL0D |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 31.15.3. Parameters

None

### 31.15.4. Examples of Use

The use of operator PulseCounter is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

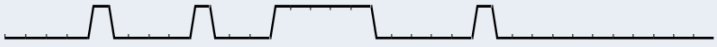
  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

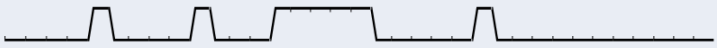
# 31.16. Operator RsFlipFlop

**Operator Library: Signal**

The operator implements a set-reset flip-flop. If a one (HIGH) is at input link Set, the operator sets the output value to HIGH and keeps the value until a zero (LOW) is present at input link Reset. When Reset is set to HIGH, the output value on output link O is set to LOW. Parameter *Priority* is used to define if Set or Reset is the master. The value defined with parameter *Init* is output at startup. See the following truth table and waveform for explanation.

| Set | Reset | O |
|---|---|---|
| 0 | 0 | O(t-1) |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 (Priority = Set) |

| Type | Name | |
|---|---|---|
| Parameter | Priority | RESET |
| Parameter | Init | 0 |
| Input | Set |  |
| Input | Reset |  |
| Output | O |  |

| Type | Name | |
|---|---|---|
| Parameter | Priority | SET |
| Parameter | Init | 1 |
| Input | Set |  |
| Input | Reset |  |
| Output | O |  |

This operator is excluded from the VisualApplets functional simulation.

## 31.16.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | Set, signal data input<br>Reset, signal data input |
| **Output Link** | O, signal data output |

## 31.16.2. Supported Link Format

| Link Parameter | Input Link Set | Input Link Reset | Output Link O |
|---|---|---|---|
| **Bit Width** | 1 | 1 | as Set |

| Link Parameter | Input Link Set | Input Link Reset | Output Link O |
|---|---|---|---|
| **Arithmetic** | unsigned | unsigned | as Set |
| **Parallelism** | 1 | 1 | as Set |
| **Kernel Columns** | 1 | 1 | as Set |
| **Kernel Rows** | 1 | 1 | as Set |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL | as Set |
| **Color Format** | VAF_GRAY | VAF_GRAY | as Set |
| **Color Flavor** | FL_NONE | FL_NONE | as Set |
| **Max. Img Width** | any | any | as Set |
| **Max. Img Height** | any | any | as Set |

## 31.16.3. Parameters

| **Priority** | |
|---|---|
| **Type** | static parameter |
| **Default** | RESET |
| **Range** | {RESET, SET} |
| This parameter specifies behavior of the operator for the case when both Set and Reset are set to HIGH simultaneously. RESET: the value on the link O will be set to LOW. SET: the value on the output link O will be set to HIGH. | |

| **Init** | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| This parameter specifies the start up value. | |

## 31.16.4. Examples of Use

The use of operator RsFlipFlop is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

# 31.17. Operator RxSignalLink

**Operator Library: Signal**

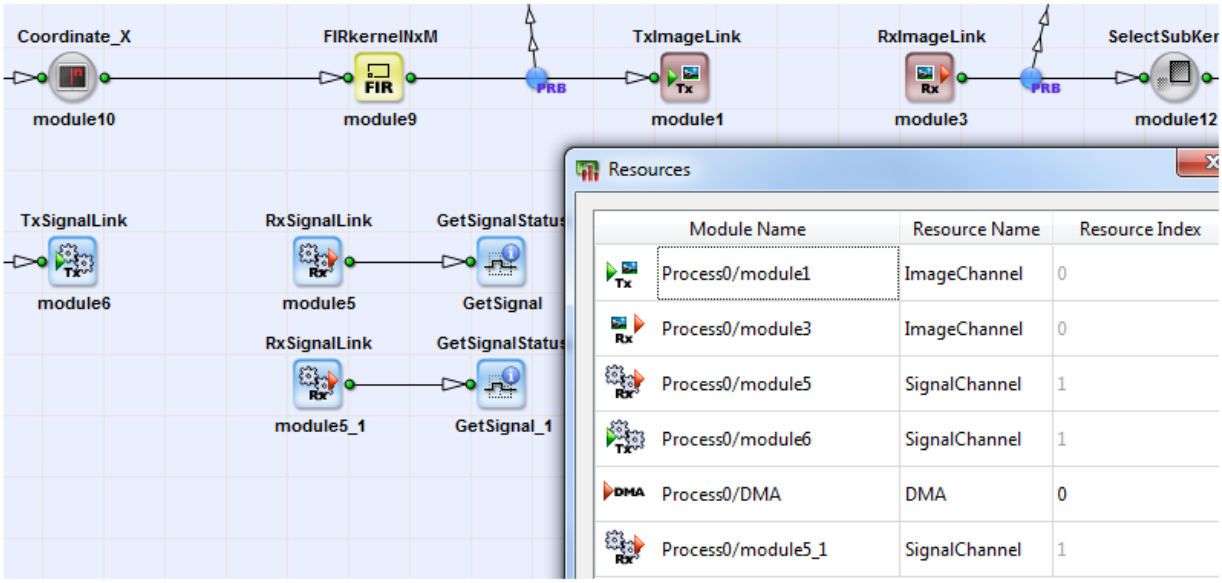
Operator *RxSignalLink* is used to receive signal data from a *TxSignalLink* operator in the same design.

Both operators establish a connection without a link. This is useful for inter-process communication, i.e., for connections between different hierarchical boxes without the need to draw a link and feedbacks.

The parameter *Channel_ID* defines a channel ID to address the sending *TxSignalLink* operator. The parameter value has to match with the ID of one of the *TxSignalLink* operators in the same design. Multiple *RxSignalLink* operators in the design may use the same channel ID.

Each *RxSignalLink* operator in a design is connected to exactly one *TxSignalLink* operator via one channel ID. In the *Resource Dialog* of VisualApplets, you can see that each *TxSignalLink* operator uses one resource *SignalChannel* exclusively. Resource *SignalChannel* allows to control the assignment of individual *TxSignalLink* operators to one or multiple *RxSignalLink* operators.

This operator is excluded from the VisualApplets functional simulation.




## 31.17.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, signal output |

## 31.17.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |

| Link Parameter | Output Link O |
|---|---|
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 31.17.3. Parameters

| Channel_ID | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 1023] |
| The channel ID of the signal link. See descriptions above. | |

### 31.17.4. Examples of Use

The use of operator RxSignalLink is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 13.3, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

# 31.18. Operator Select

**Operator Library: Signal**

The operator Select implements a N-input to 1-output multiplexer controlled by the *Select* parameter. N must be smaller or equal to 64.

This operator is excluded from the VisualApplets functional simulation.

## 31.18.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I[n], signal data input |
| **Output Link** | O, signal data output |

## 31.18.2. Supported Link Format

| Link Parameter | Input Link I[n] | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I0 |
| **Arithmetic** | unsigned | as I0 |
| **Parallelism** | 1 | as I0 |
| **Kernel Columns** | 1 | as I0 |
| **Kernel Rows** | 1 | as I0 |
| **Img Protocol** | VALT_SIGNAL | as I0 |
| **Color Format** | VAF_GRAY | as I0 |
| **Color Flavor** | FL_NONE | as I0 |
| **Max. Img Width** | any | as I0 |
| **Max. Img Height** | any | as I0 |

## 31.18.3. Parameters

| **Select** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, N-1] |
| This parameter is used to select an input. | |

## 31.18.4. Examples of Use

The use of operator Select is shown in the following examples:

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection

to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

  Examples - Demonstration of how to use the operator

# 31.19. Operator SetSignalStatus

**Operator Library: Signal**

The operator's signal link output can be set by software using parameter *Mode*. The state type is specified by the parameter Mode. Three options are available:

- **`Low`**:

  The operator output signal link has low value (zero).

- **`High`**:

  The operator output signal link has high value (one).

- **`Pulse`**:

  The operator will generate a pulse at its output, i.e. will output a high value for one clock cycle. After this pulse, the operator will output low value (zero). Setting the parameter to **`Pulse`** after it was set to **`High`** will not generate a low value before the pulse. Hence, the parameter should be set to **`Low`** or **`Pulse`** before setting it to **`Pulse`**.

This operator is excluded from the VisualApplets functional simulation.

## 31.19.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Output Link** | O, signal data output |

## 31.19.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 31.19.3. Parameters

| **Mode** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | Low |
| **Range** | {Low, High, Pulse} |
| This parameter specifies the output signal value at the output link of the operator. See description above. | |

### 31.19.4. Examples of Use

The use of operator SetSignalStatus is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.6.7, 'Image Flow Control'

  Example - For debugging purposes of the designs internal data flow control in hardware and a possible compensation.

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 31.20. Operator ShaftEncoder

**Operator Library: Signal**

The operator analyzes two shaft encoder signal traces. The two traces are fed into the operator. At the outputs of the operator, a pulse for every encoder line as well as a direction signal are provided. The phase between the two trace signals are used for detection.

For every detected encoder line, the operator generates a pulse at its output link O. The pulse with is one design clock cycle. At output link Dir, the detected direction of the encoder is outputs. If Dir = 0 (LOW), a forward direction was detected. If Dir = 1 (HIGH), a reverse direction was detected. The values at the Dir output are only valid when output O is 1 (HIGH) i.e. a pulse is output.

Parameter *LeadingTrace* is used to define which one of the two traces is the leading trace, i.e. which trace is 90° ahead.

Parameter Mode defines the detection mode. The following modes can be used:

- **ModeX1**:

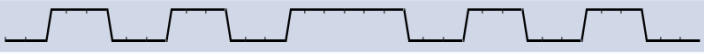
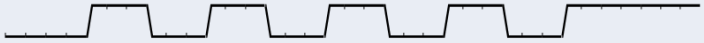
  Single Speed Mode. Only one edge of the leading signal is used for detection. The output is determined by:

$$O = L_R \wedge \overline{S} \vee L_F \wedge \overline{S}$$

$$Dir = L_F \wedge \overline{S}$$

  where $L_R$ and $L_F$ are the rising and falling edges of the leading trace e.g. A. $S$ is the slave trace e.g. B.

  The following figure illustrates the behavior of the ModeX1

| Type | Name | |
|---|---|---|
| Parameter | LeadingTrace | A |
| Parameter | Init | Mode1X |
| Input | A |  |
| Input | B |  |
| Output | O |  |
| Output | Dir |  |

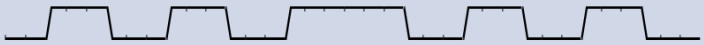
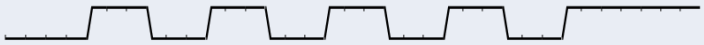
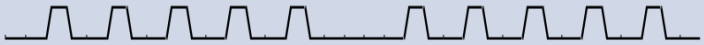
- **ModeX2**:

  Double Speed Mode. Both edges of the leading signal are used for detection. The output is determined by:

$$O = L_R \vee L_F$$

$$Dir = L_F \wedge \overline{S} \vee L_R \wedge S$$

  The following figure illustrates the behavior of the ModeX2

| Type | Name | |
|---|---|---|
| Parameter | LeadingTrace | A |
| Parameter | Init | Mode2X |
| Input | A |  |
| Input | B |  |
| Output | O |  |
| Output | Dir |  |

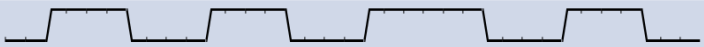
- **ModeX4**:

  Quad Speed Mode. All edges of the leading and the slave signal are used for detection. The output is determined by:

$$O = L_R \vee L_F\, S_R \vee S_F$$

$$Dir = L_F \wedge \overline{S} \vee L_R \wedge S \vee S_R \wedge \overline{L} \vee S_F \wedge L$$

  The following figure illustrates the behavior of the ModeX4

| Type | Name | |
|---|---|---|
| Parameter | LeadingTrace | A |
| Parameter | Init | Mode4X |
| Input | A |  |
| Input | B |  |
| Output | O |  |
| Output | Dir |  |

An additional reset input can be used to reset the operator to its initial state.

This operator is excluded from the VisualApplets functional simulation.

## 31.20.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | TraceA, signal data input<br>TraceB, signal data input<br>Reset, signal data input |
| **Output Links** | O, signal data output of detected encoder lines<br>Dir, signal data output of detected direction |

## 31.20.2. Supported Link Format

| Link Parameter | Input Link TraceA | Input Link TraceB | Input Link Reset |
|---|---|---|---|
| **Bit Width** | 1 | 1 | 1 |

| Link Parameter | Input Link TraceA | Input Link TraceB | Input Link Reset |
|---|---|---|---|
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any | any |
| **Max. Img Height** | any | any | any |

| Link Parameter | Output Link O | Output Link Dir |
|---|---|---|
| **Bit Width** | as TraceA | as TraceA |
| **Arithmetic** | as TraceA | as TraceA |
| **Parallelism** | as TraceA | as TraceA |
| **Kernel Columns** | as TraceA | as TraceA |
| **Kernel Rows** | as TraceA | as TraceA |
| **Img Protocol** | as TraceA | as TraceA |
| **Color Format** | as TraceA | as TraceA |
| **Color Flavor** | as TraceA | as TraceA |
| **Max. Img Width** | as TraceA | as TraceA |
| **Max. Img Height** | as TraceA | as TraceA |

### 31.20.3. Parameters

| **LeadingTrace** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | A |
| **Range** | {A, B} |
| This parameter specifies the leading trace. The leading trace is 90° ahead. | |

| **Mode** | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | Mode1X |
| **Range** | {Mode1X, Mode2X, Mode4X} |
| This parameter defines the detection mode of the shaft encoder. See description above. | |

### 31.20.4. Examples of Use

The use of operator ShaftEncoder is shown in the following examples:

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 31.21. Operator ShaftEncoderCompensate

**Operator Library: Signal**

The operator compensates runbacks of a ShaftEncoder.

The operator has an input link I which can be fed with encoder pulses On input link *Dir*, the current encoder direction can be defined.

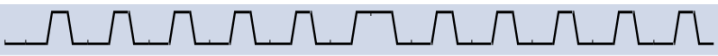
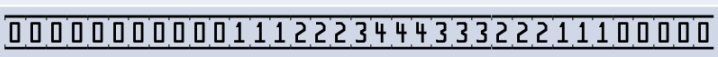
Input pulses with backward direction are suppressed at the output. However, the operator will count the input pulses with backward direction internally. When the ShaftEncoderCompensate is fed with forward pulses again, they will be still suppressed until all backward pulses are compensated. In other words, each backward input pulse increments the internal counter. Each forward input pulse decreases this counter if it is > 0. A pulse is only output for each forward input pulse if the internal counter is equal to zero.

The parameter *MaxPulseBits* specifies the bit width of the internal counter, i.e. ShaftEncoderCompensate can compensate 2 ^ MaxPulseBits reverse pulses at maximum. If more than 2 ^ MaxPulseBits backward pulses occur, only 2 ^ MaxPulseBits pulses during runbacks can be compensated.

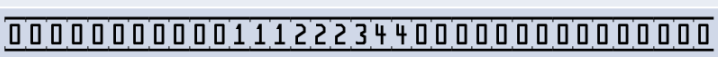
The current counter value i.e. the number of pulses to be compensated can be read using parameter *CompensationCounter*. It is possible to overwrite the current counter value by writing a new value to this parameter.

The operator can directly be connected to operator *ShaftEncoder*.

In the following waveform the functionality of the operator is illustrated.

| Type | Name | |
|---|---|---|
| Input | I |  |
| Input | Dir |  |
| Input | Reset |  |
| Output | O |  |
| Parameter | Compensation Counter |  |

An additional reset input can be used to reset the counter as shown in the following.

| Type | Name | |
|---|---|---|
| Input | I |  |
| Input | Dir |  |
| Input | Reset |  |
| Output | O |  |
| Parameter | Compensation Counter |  |

This operator is excluded from the VisualApplets functional simulation.

### 31.21.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, signal data input<br>Dir, signal data input<br>Reset, signal data input |
| **Output Link** | O, signal data output |

### 31.21.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Dir |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 31.21.3. Parameters

| **MaxPulseBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 10 |
| **Range** | [2, 32] |
| This parameter specifies the bit width of the internal counter. | |

| **CompensationCounter** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, 2^MaxPulseBits-1] |

| CompensationCounter |
| --- |
| This parameter shows the current value of the compensation counter. Reading from the parameter shows the current value. Writing to the parameter overwrites the counter value. |

## 31.21.4. Examples of Use

The use of operator ShaftEncoderCompensate is shown in the following examples:

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

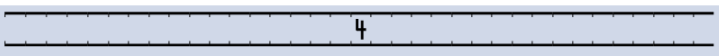
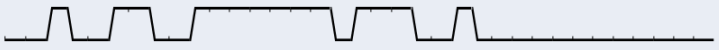
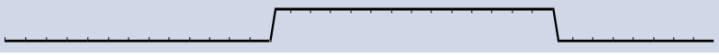
- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

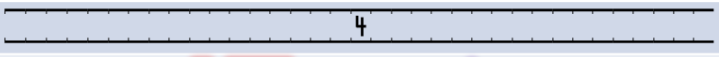
# 31.22. Operator SignalDebounce

**Operator Library: Signal**

The operator suppresses fast signal changes. Parameter *Debounce* defines a minimum time an input signal has to be constant before it is forwarded to the output. Inputs which are changing faster are suppressed. The following figure illustrates the operator's behavior.

| Type | Name | |
|---|---|---|
| Parameter | Debounce |  |
| Input | I |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O |  |

The debounce time is measured in Ticks being one (HIGH), i.e. Tick acts like a prescaler input. Is Tick set constantly to one (HIGH), the debounce operator counts the signal duration at the design clock frequency. In most applications the Tick input is not required. Tie it to operator Vcc in these cases. The following waveform shows the operator's behavior when using the Tick input link.

| Type | Name | |
|---|---|---|
| Parameter | Debounce |  |
| Input | I |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O |  |

Note that the operator acts like a delay element. Any signal changes that meet the debounce specification of being unchanged for N Ticks will be forwarded to the operator output after the delay of N Ticks. However, the pulse form remains unchanged.

An additional reset input allows the reset of the operator.

This operator is excluded from the VisualApplets functional simulation.

## 31.22.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, signal data input<br>Tick, signal data input<br>Reset, signal data input |
| **Output Link** | O, signal data output |

## 31.22.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Bit Width** | 1 | 1 |

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 31.22.3. Parameters

| **DebounceBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1, 64] |
| This parameter specifies the bit maximum range of parameter *Debounce*. Only of *Debounce* set to dynamic, this parameter can be changed. | |

| **Debounce** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 2 |
| **Range** | [0, 2^DebounceBits-1] if dynamic, [0, 2^64-1] if static |
| Defines the debounce time. | |

### 31.22.4. Examples of Use

The use of operator SignalDebounce is shown in the following examples:

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

  Examples - Demonstration of how to use the operator

# 31.23. Operator SignalDelay

**Operator Library: Signal**

The operator delays the input signal and provides it on its output. The delay is controlled by parameter *Delay*.

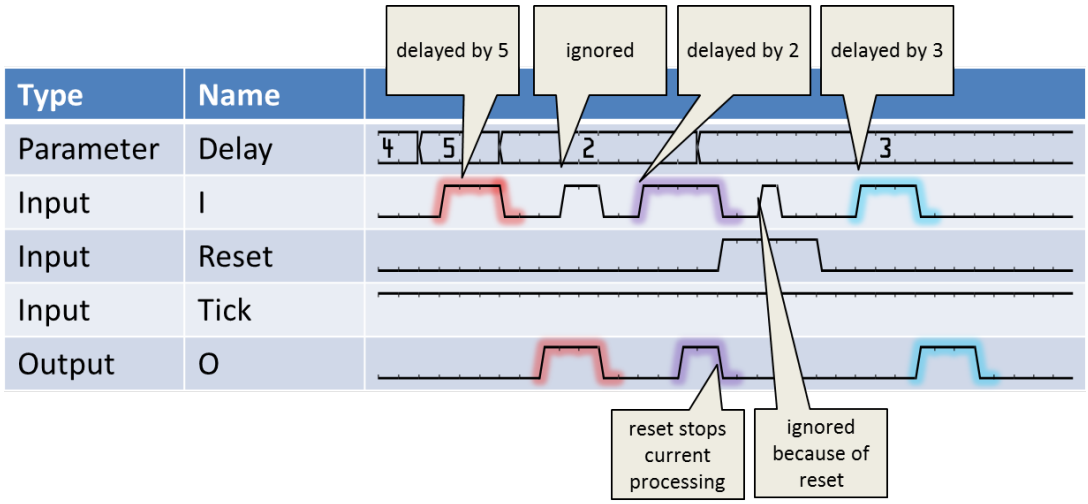
The rising edge of a pulse at the input starts the delay. The current value of parameter *Delay* defines the delay. The pulse width of the input pulse is kept i.e. the rising and falling edges are delayed.

The operator can be run in two modes defined using parameter *Mode*:

- SinglePulse:

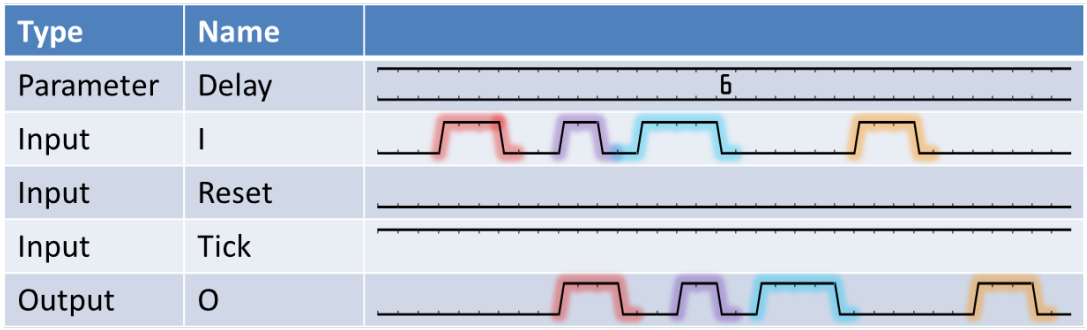
  The operator can delay a single pulse at once only. During delaying of a pulse no new pulses at the operator input can be accepted. In this case, every new input pulse will be ignored. Any changes of the delay parameter are ignored while a pulse is currently processed.

  The operator can be reseted using input link Reset. While the reset input is high, no output pulses are processed. Any processing is aborted. The operator restarts operation when the reset input is low. The following waveform illustrates the operator's behavior.




- MultiplePulses:

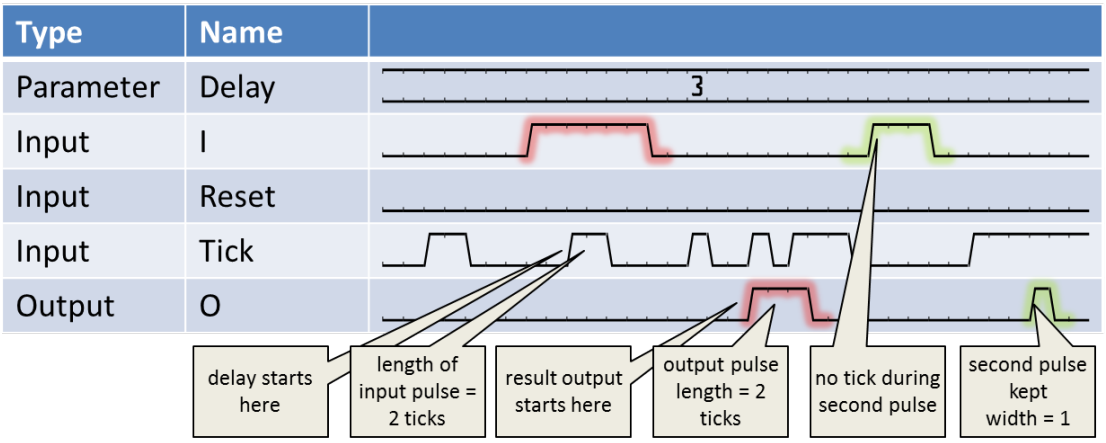
  In this mode, the operator can delay multiple input pulses at the same time. The operator is a shift register of fixed length. The delay cannot be changed in this mode. In the following figure, the mode is illustrated.




  The reset input can result in undefined values if used in combination with the MutliPulses mode. Therefore, the reset input must be set for the duration defined in the parameter *DurationBits* if used. Tie it to GND if not used.

The intention of the MultiPulses mode is to use it for short delays only. Long delays will require many resources.

The Tick input defines the time, the operator is processing data. It can be used like a prescaler. In most cases, the Tick input is not required. In this case, tie it to operator VCC. In the following figure, the influence of the Tick input is shown.




One special case when using ticks is that input pulses are sampled even if no tick is present. This is shown for the second input pulse of the waveform. This ensures that no input pulses can get lost.

This operator is excluded from the VisualApplets functional simulation.

### 31.23.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, signal data input<br>Tick, signal data input<br>Reset, signal data input |
| **Output Link** | O, signal data output |

### 31.23.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 31.23.3. Parameters

| **Mode** | |
|---|---|
| **Type** | static parameter |
| **Default** | SinglePulse |
| **Range** | {SinglePulse, MultiplePulses} |
| Defines if the operator can delay only a single or multiple pulses at the same time. See description above. | |

| **DelayBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1, 64] |
| The maximum possible signal delay is defined using this parameter. This parameter is enabled only if *Mode* is set to SinglePulse. | |

| **Delay** | |
|---|---|
| **Type** | dynamic/static parameter |
| **Default** | 0 |
| **Range** | [0,2^DelayBits-1] if Mode=SinglePulse, [1, 2^64-1] else |
| The actual delay is defined using this parameter. The parameter is always static if *Mode* is set to MultiplePulses. | |

### 31.23.4. Examples of Use

The use of operator SignalDelay is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

  Examples - Demonstration of how to use the operator

# 31.24. Operator SignalEdge

**Operator Library: Signal**

The operator SignalEdge generates a pulse of one design clock cycle when it detects a level changing of the input signal. The parameter *Edge* specifies the type of the detecting edge. Three settings are possible:

- **`Rising`**:

  A pulse for each rising edge at the input is generated.

| Type | Name | |
|---|---|---|
| Parameter | Edge | `RisingEdge` |
| Input | I |  |
| Output | O |  |

- **`Falling`**:

  A pulse for each falling edge at the input is generated.

| Type | Name | |
|---|---|---|
| Parameter | Edge | `FallingEdge` |
| Input | I |  |
| Output | O |  |

- **`Both`**:

  A pulse for both, rising and falling edges at the input is generated.

| Type | Name | |
|---|---|---|
| Parameter | Edge | `BothEdges` |
| Input | I |  |
| Output | O |  |

This operator is excluded from the VisualApplets functional simulation.

## 31.24.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, signal data input |
| **Output Link** | O, signal data output |

### 31.24.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 31.24.3. Parameters

| **Edge** | |
|---|---|
| **Type** | static parameter |
| **Default** | RisingEdge |
| **Range** | {RisingEdge, FallingEdge, BothEdges} |
| The parameter defines the type of edge the operator detects. | |

### 31.24.4. Examples of Use

The use of operator SignalEdge is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

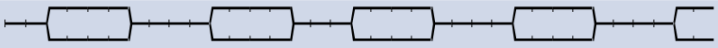
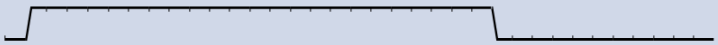
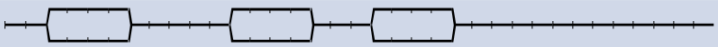
  Examples - Demonstration of how to use the operator

# 31.25. Operator SignalGate

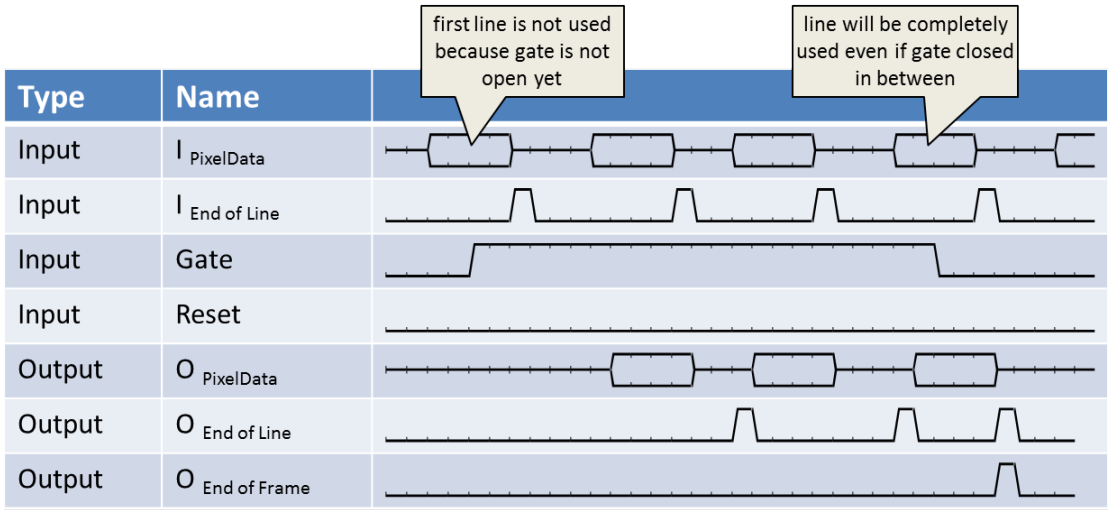
**Operator Library: Signal**

The operator gates the image stream between the input I and the output O. The gate is controlled by the signal input link Gate. If the gate is closed, the input data is discarded. If the gate is open, pixels are forwarded to the output. However, the operator ensures that the integrity of lines or frames is not destroyed i.e. always full lines or frames are transfered and will not be cut. If the operator is used with the input image protocol VALT_LINE1D, the operator will assemble the lines within one open gate period into a frame. Thus it combines lines and converts an infinite line stream for example of a line scan camera into frames. The operator is often used as image trigger for line scan cameras.

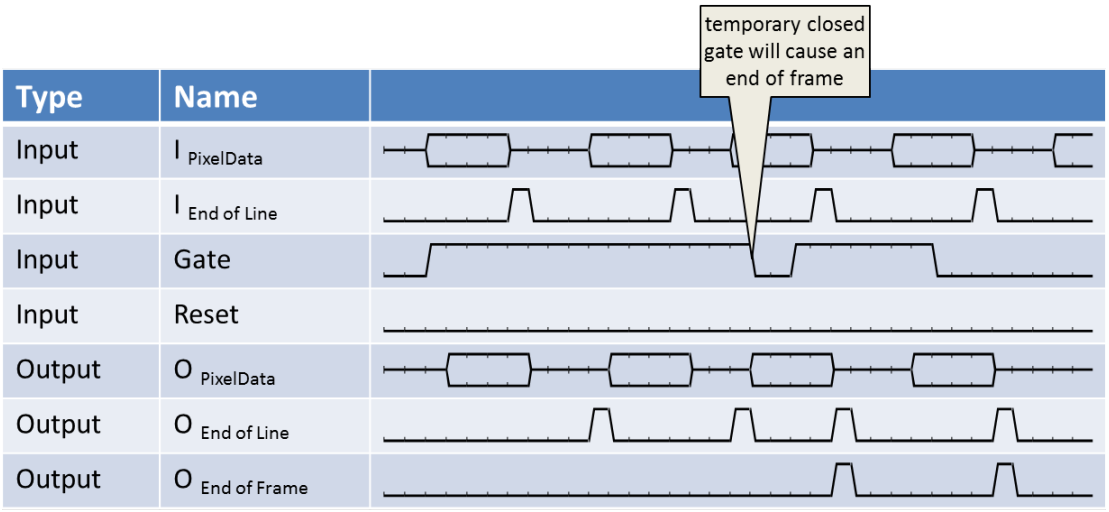
In the following figure an example of the operator's behavior is shown. As you can see, lines are fed into the operator. Each line is terminated with an end of line control signal which is embedded in the image link stream. The gate is open for a duration of three lines. At the output the first three lines are present while the fourth and fifth are discarded. Moreover, after the last line before the gate is closed, an end of frame marker is added to the image data stream. To summarize, in the example, the operator assembled the first three lines into a frame. Data is discarded during the closed periods.

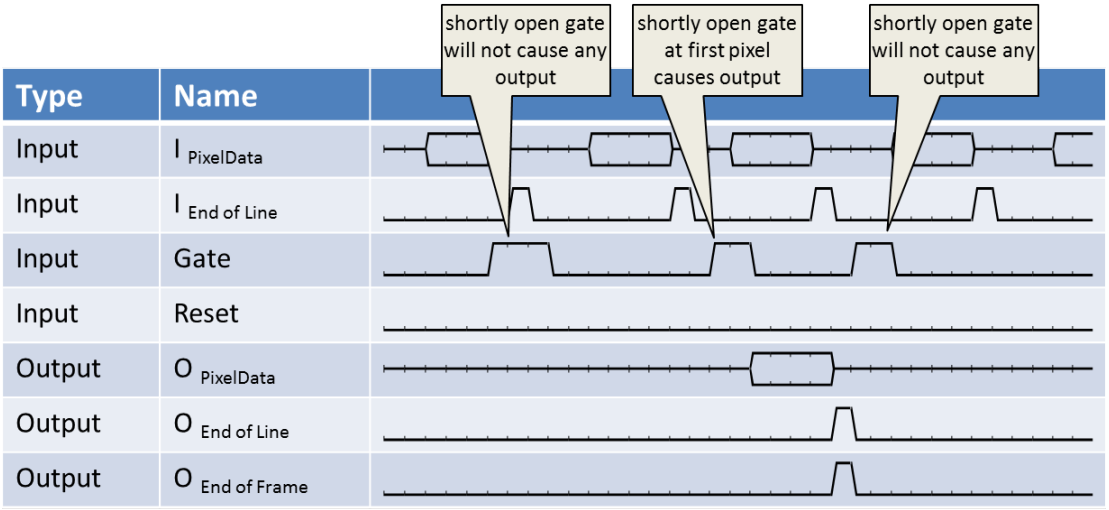
| Type | Name | |
|---|---|---|
| Input | I PixelData |  |
| Input | I End of Line |  |
| Input | Gate |  |
| Input | Reset |  |
| Output | O PixelData |  |
| Output | O End of Line |  |
| Output | O End of Frame |  |

If the gate opens while a line is currently processed, the line will not be used. Moreover, if the gate closes while a line is currently processed, the line will be fully transfered to the output. This ensures the integrity of lines. The following waveform shows these cases.



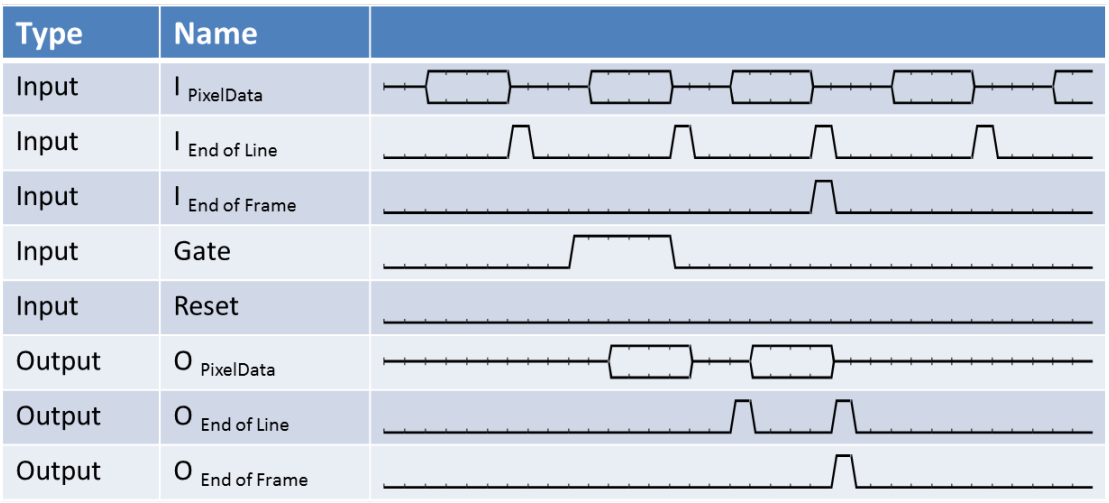

A shortly closed gate will cause the generation of an end of frame. The next two waveforms show short periods of open or closed gates.




**Keep in mind, a line will be used if the gate is open at the same time as the first pixel of a line is present at the input link. An end of frame is generated after a line if the respective line is not discarded and falling edge of the gate input was present after the first line pixel.**

Instead of a VALT_LINE1D line data stream, the operator can be used with a VALT_IMAGE2D image data stream at input I. In this case, the operator will forward the full input frame if the gate is open simultaneously to the first pixel of the image as can be seen in the following.



The reset input can be used to reset the operator i.e. to cut a line, and restart.



**Simplified Waveforms**

The waveform illustrations are simplified to show the exact operator's behavior. The real implementation delays the end of line and end of frame output.



**Operator Violates Max.Image Height**

If the gate is constantly open, the operator will generate an image of a large height. If this height exceeds the Max. Image Height link property set for the output link, the VisualApplets rules are violated. You should either cut exceeding images using operator *SplitImage* or delete exceeding lines (*RemoveLine*, *SelectROI*).

This operator is excluded from the VisualApplets functional simulation.

## 31.25.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I, image data input<br>Gate, signal input<br>Reset, signal input |
| **Output Link** | O, image data output |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.
- Input link *I* is asynchronous to the signal inputs.

## 31.25.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Gate |
|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | 1 |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | VALT_SIGNAL |
| **Color Format** | any | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | any if input VALT_IMAGE1D, else as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

## 31.25.3. Parameters

None

## 31.25.4. Examples of Use

The use of operator SignalGate is shown in the following examples:

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

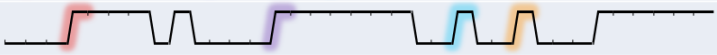
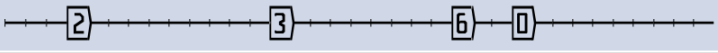
- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

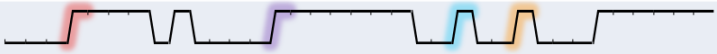
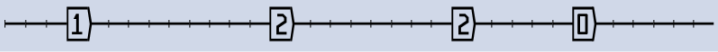
# 31.26. Operator SignalToDelay

**Operator Library: Signal**

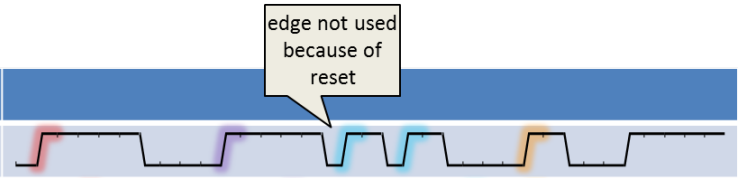
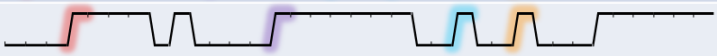
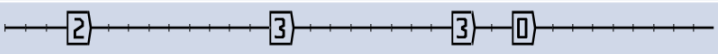
The operators measures the delay between the rising edge of the signal at input link I0 and the rising edge of the signal at input link I1. The result is output as a pixel value at output link O. Hence, output O is a VALT_PIXEL0D pixel data stream. The bit width of the output link can be changed and represents the maximum possible delay. If the actual delay exceeds the output value range, the delay will be clipped to the maximum possible value. In the following waveform, the behavior of the operator is illustrated.

| Type | Name | |
|---|---|---|
| Input | I1 |  |
| Input | I2 |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O PixelData |   |

The delay is measured in Ticks being high. Tick is a signal input and can be used like a prescaler. For every high value at the Tick input, the delay time is measured. The following waveform shows the behavior of the Tick input to the delay measurement. In most cases, the Tick input is not required. Tie it to operator VCC in this case.

| Type | Name | |
|---|---|---|
| Input | I1 |  |
| Input | I2 |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O PixelData |   |

By use of the additional reset input, the current measurement can be cancelled and the operator will accept a new rising edge at input I0.

| Type | Name | |
|---|---|---|
| Input | I1 |   |
| Input | I2 |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O PixelData |   |

This operator is excluded from the VisualApplets functional simulation.

### 31.26.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I0, signal input<br>I1, signal input<br>Tick, signal input<br>Reset, signal input |
| **Output Link** | O, image data output |

### 31.26.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1 | Input Link Tick |
|---|---|---|---|
| **Bit Width** | 1 | 1 | 1 |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any | any |
| **Max. Img Height** | any | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | [1, 64] |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_PIXEL0D |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 31.26.3. Parameters

None

### 31.26.4. Examples of Use

The use of operator SignalToDelay is shown in the following examples:

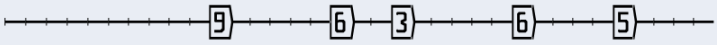
- Section 13.3, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

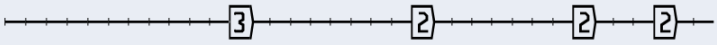
# 31.27. Operator SignalToPeriod

**Operator Library: Signal**

The operators measures the the period time of the signal at the input. The period time is the time between two rising edges at the input. The result is output as a pixel value at output link O. Hence, output O is a VALT_PIXEL0D pixel data stream. The bit width of the output link can be changed and represents the maximum possible period time. If the actual period time exceeds the output value range, it will be clipped to the maximum possible value. In the following waveform, the behavior of the operator is illustrated.

| Type | Name | |
|---|---|---|
| Input | I |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O PixelData |   |

The period time is measured in Ticks being high. Tick is a signal input and can be used like a prescaler. For every high value at the Tick input, the period time is measured. The following waveform shows the behavior of the Tick input to the period measurement. In most cases, the Tick input is not required. Tie it to operator VCC in this case.

| Type | Name | |
|---|---|---|
| Input | I |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O PixelData |   |

By use of the additional reset input, the current measurement can be cancelled.

This operator is excluded from the VisualApplets functional simulation.

## 31.27.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, signal input<br>Tick, signal input<br>Reset, signal input |
| **Output Link** | O, image data output |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.

### 31.27.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | [1, 64] |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_PIXEL0D |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 31.27.3. Parameters

None

### 31.27.4. Examples of Use

The use of operator SignalToPeriod is shown in the following examples:

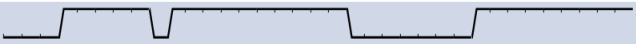
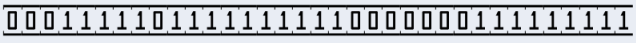
- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

  Examples - Demonstration of how to use the operator

# 31.28. Operator SignalToPixel

**Operator Library: Signal**

The operator converts up to 64 input signal streams into a VALT_PIXEL0D pixel data stream. The current signal value of an input is converted into a pixel bit: LOW is converted to value 0 and HIGH is converted to value 1. If there is more than a single input port then each input maps to a bit position of the output pixel data. In particular the first input port I000 maps to bit 0 of the output port O and subsequent input ports map to subsequent bit positions of output port O.

| Type | Name | |
|---|---|---|
| Input | I |  |
| Output | O |  |

A steady pixel stream is generated, i.e., one pixel is output for every clock cycle. This is because signals are defined for every clock cycle while pixel data streams may have gaps between the pixels. Since the signal stream has to be converted into a pixel stream, a pixel has to be output for every clock cycle, too. This can cause data lost. Pixel processing operators may block their inputs. However, the output of operator PixelToSignal can't be blocked as pixel always occur. Increase the parallelism or reduce the number of pixels to avoid data lost.

This operator is excluded from the VisualApplets functional simulation.

## 31.28.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I000...I063, signal input |
| **Output Link** | O, image data output |

## 31.28.2. Supported Link Format

| Link Parameter | Input Link I000...I063 | Output Link O |
|---|---|---|
| **Bit Width** | 1 | [1, 64] |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_PIXEL0D |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

## 31.28.3. Parameters

None

## 31.28.4. Examples of Use

The use of operator SignalToPixel is shown in the following examples:

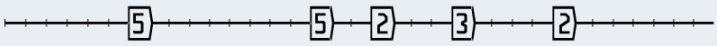
- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

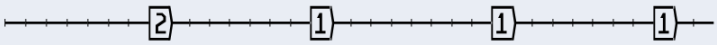
# 31.29. Operator SignalToWidth

**Operator Library: Signal**

The operators measures the pulse width of the signal at the input. The pulse width the time between a rising and falling edge at the input. The result is output as a pixel value at output link O. Hence, output O is a VALT_PIXEL0D pixel data stream. The bit width of the output link can be changed and represents the maximum possible pulse width. If the actual pulse width exceeds the output value range, it will be clipped to the maximum possible value. In the following waveform, the behavior of the operator is illustrated.

| Type | Name | |
|---|---|---|
| Input | I |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O PixelData |   |

The period time is measured in Ticks being high. Tick is a signal input and can be used like a prescaler. For every high value at the Tick input, the period time is measured. The following waveform shows the behavior of the Tick input to the period measurement. In most cases, the Tick input is not required. Tie it to operator VCC in this case.

| Type | Name | |
|---|---|---|
| Input | I |  |
| Input | Tick |  |
| Input | Reset |  |
| Output | O PixelData |   |

An input pulse between two tick pulses will not get lost. The operator remembers the pulse an will output the result = 1 with the next tick. If more than one pulse is between two ticks, the pulse will be treated as a single pulse. The input is under-sampled in this case.

By use of the additional reset input, the current measurement can be cancelled.

This operator is excluded from the VisualApplets functional simulation.

## 31.29.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, signal input<br>Tick, signal input<br>Reset, signal input |
| **Output Link** | O, image data output |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.

### 31.29.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | [1, 64] |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_PIXEL0D |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

### 31.29.3. Parameters

None

### 31.29.4. Examples of Use

The use of operator SignalToWidth is shown in the following examples:

- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

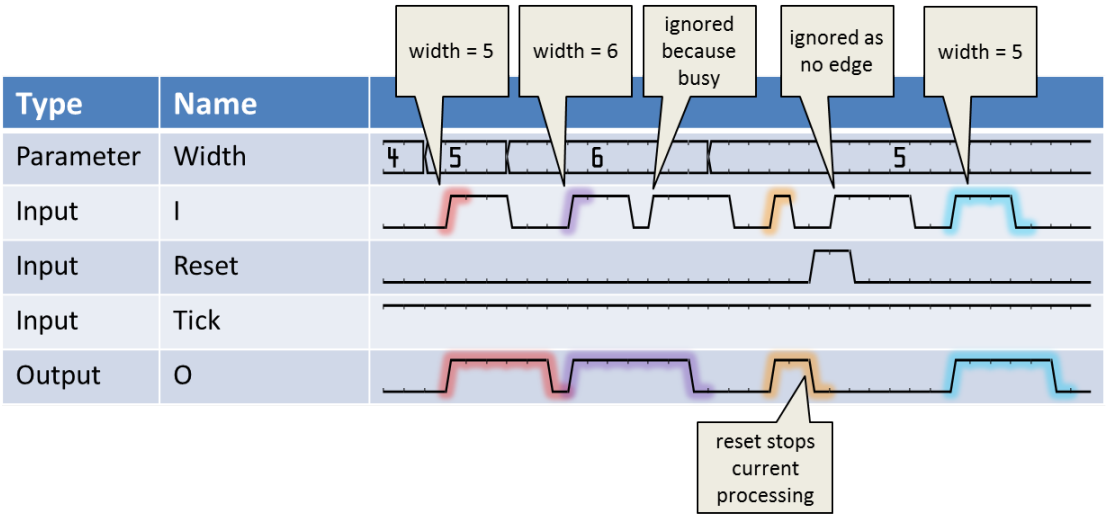
  Examples - Demonstration of how to use the operator

# 31.30. Operator SignalWidth

**Operator Library: Signal**

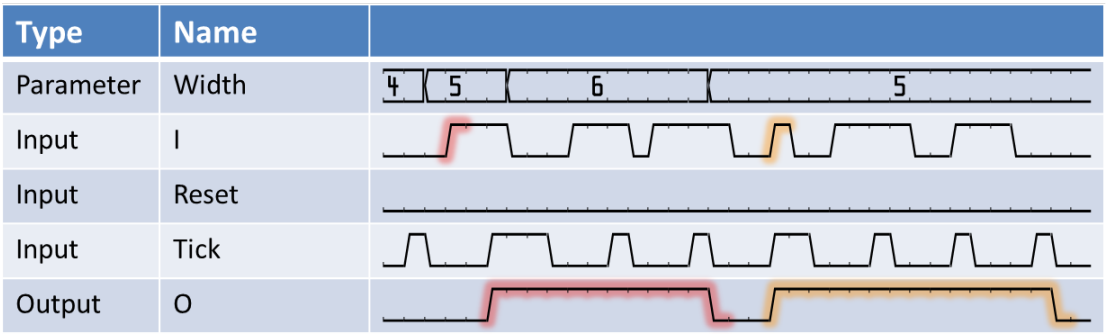
The operator generates a pulse with controllable width. The width is defined using parameter *Width*.

An output pulse is generated for rising edges at the input. If a a pulse is currently generated, no more rising edges at the input can be accepted. See the following waveform for explanations.




Via the input link Reset the user can reset the operator to its initial state.

The Tick input defines the time, the operator is processing data. It can be used like a prescaler. In most cases, the Tick input is not required. In this case, tie it to operator VCC. In the following figure, the influence of the Tick input is shown.




One special case when using ticks is that input pulses are sampled even if no tick is present. This is shown for the first input pulse of the waveform. This behavior ensures that no input pulses can get lost.

Pulses between two tick pulses are not lost (as long as it is only one pulse, i.e., as long as the pause between two incoming pulses is longer than a tick period). The operator recognizes the puls and starts interpretation of the puls as soon as the next tick is HIGH. Those small pulses are adapted to the tick time base and synchronized to the ticks. All changes at the operator output take place when tick is HIGH. Only if multiple input pulses come in between two occurences of tick = HIGH, the operator is unable to recognize them as different pulses as they are below the tick time base.

This operator is excluded from the VisualApplets functional simulation.

## 31.30.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |

| Property | Value |
|---|---|
| **Input Links** | I, signal input<br>Tick, signal input<br>Reset, signal input |
| **Output Link** | O, signal output |

## 31.30.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Tick |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | asI |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 31.30.3. Parameters

| WidthBits | |
|---|---|
| **Type** | static parameter |
| **Default** | 16 |
| **Range** | [1; 64] |

The maximum possible width is defined using this parameter. Stepsize is 1. This parameter is enabled only if *Width* is set to dynamic.

If you reduce the value of paramter WidthBits so that the value of Width is bigger than the new maximal value, both parameters are displayed in red. The DRC shows an according error message.

| Width | |
|---|---|
| **Type** | dynamic/static parameter |
| **Default** | 32768 |
| **Range** | [0;2^WidthBits-1] if dynamic, [0; 2^64-1] if static |

| Width |
| --- |
| The actual output pulse width is defined using this parameter.<br><br>Stepsize is 1 for dynamic and static.<br><br>When width=0 there is no output.<br><br>If set to dynamic, parameter WidthBits is activated. If set to static, parameter WidthBits is de-activated.<br><br>If you reduce the value of paramter WidthBits so that the value of Width is bigger than the new maximal value, both parameters are displayed in red. The DRC shows an according error message. |

## 31.30.4. Examples of Use

The use of operator SignalWidth is shown in the following examples:

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.6, 'Area Scan Trigger for imaFlex CXP-12 Quad'

  An area scan trigger for CoaXPress12 is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

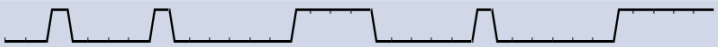
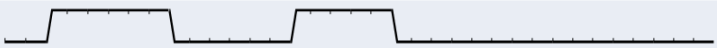
- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

  Examples - Demonstration of how to use the operator

# 31.31. Operator SyncSignal

**Operator Library: Signal**

The operator synchronizes N input links to the master signal and provides the synchronized version of the inputs at the outputs. For each rising edge at input link MasterI, the inputs I0..In are sampled. Thus the operator includes N registers, enabled with the rising edges of MasterI input. See the following waveform for illustration.

| Type | Name | |
|---|---|---|
| Input | MasterI |  |
| Input | I0 |  |
| Input | I1 |  |
| Input | Reset |  |
| Output | O0 |  |
| Output | O1 |  |

The additional reset input link allows the reset of the operator. All outputs are set to zero, while reset.

This operator is excluded from the VisualApplets functional simulation.

## 31.31.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | MasterI, signal input<br>I0..In, signal input<br>Reset, signal input |
| **Output Link** | O0..On, signal output |

## 31.31.2. Supported Link Format

| Link Parameter | Input Link MasterI | Input Link I0..In |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Reset | Output Link O0..On |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |

| Link Parameter | Input Link Reset | Output Link O0..On |
|---|---|---|
| **Parallelism** | 1 | 1 |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | any | as MasterI |
| **Max. Img Height** | any | as MasterI |

### 31.31.3. Parameters

None

### 31.31.4. Examples of Use

The use of operator SyncSignal is shown in the following examples:

- Section 13.9, 'Functional Example for Specific Operators of Library Signal, Logic, Filter and Parameters'

  Examples - Demonstration of how to use the operator

# 31.32. Operator TxSignalLink

**Operator Library: Signal**

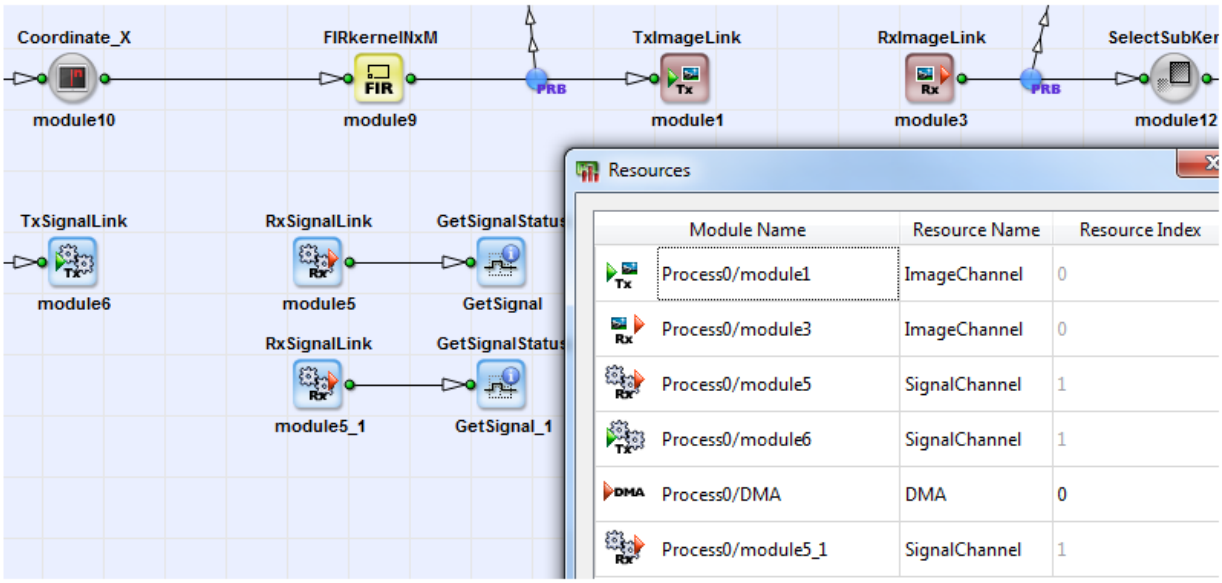
Operator *TxSignalLink* is used to send data to an *RxSignalLink* operator any place in the design. Both operators establish a connection without a link. This is useful for inter-process communication, i.e., for connections between different hierarchical boxes without the need to draw a link and feedbacks.

The parameter *Channel_ID* defines a channel ID to address the receiving *RxSignalLink* operator. The parameter value has to be unique and must not be used by any other TxSignalLink operator in the design.

There has to exist at least one *RxSignalLink* operator in the design which is using the same channel ID and will receive the signal data.

Each *TxSignalLink* operator in a design is connected to at least one *RxSignalLink* operator via one channel ID. In the *Resource Dialog* of VisualApplets, you can see that each *TxSignalLink* operator uses one resource *SignalChannel* exclusively. Resource *SignalChannel* allows to control the assignment of individual *TxSignalLink* operators to one or multiple *RxSignalLink* operators. For the number of available *SignalChannel* resources (which also defines the maximum number of allowed *TxSignalLink* operators in a design), see Appendix A, '*Device Resources*'.

This operator is excluded from the VisualApplets functional simulation.




## 31.32.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, signal input |

## 31.32.2. Supported Link Format

| Link Parameter | Input Link I |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |

| Link Parameter | Input Link I |
|---|---|
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 31.32.3. Parameters

| Channel_ID | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 1023] |
| The channel ID of the signal link. See descriptions above. | |

### 31.32.4. Examples of Use

The use of operator TxSignalLink is shown in the following examples:

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 13.4, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

# 31.33. Operator Vcc

**Operator Library: Signal**

This operator provides a signal with constant value 1 (HIGH).

This operator is excluded from the VisualApplets functional simulation.

## 31.33.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, signal output |

## 31.33.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 1 |
| **Arithmetic** | unsigned |
| **Parallelism** | 1 |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | VALT_SIGNAL |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

## 31.33.3. Parameters

None

## 31.33.4. Examples of Use

The use of operator Vcc is shown in the following examples:

- Section 12.6.3, 'Image Timing Generator'

  Example - While image timing is provided by a generator the designs data flow can be analyzed.

- Section 12.15.1.1, 'Area Scan Trigger for microEnable IV VD4-CL/-PoCL Platform Using Signal Operators and Operator **CameraControl**'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.2, 'Area Scan Trigger for microEnable IV VQ4-GE/-GPoE'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.3, 'Area Scan Trigger for microEnable 5 marathon/LightBridge VCL'

An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.4, 'Area Scan Trigger for microEnable 5 VD8-CL/-PoCL'

  An area scan trigger is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.5, 'Area Scan Trigger for microEnable 5 marathon VCX QP'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.7, 'Area Scan Trigger for microEnable 5 VQ8-CXP6B and VQ8-CXP6D'

  An area scan trigger for CoaXPress is presented. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.8.3, 'Rebuild of Operator TrgPortLine with VisualApplets Signal Processing Operators'

  A VisualApplets design example where all functions of operator *TrgPortLine* are rebuild using other signal processing operators. Allows custom trigger modules.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 31.34. Operator WidthToSignal

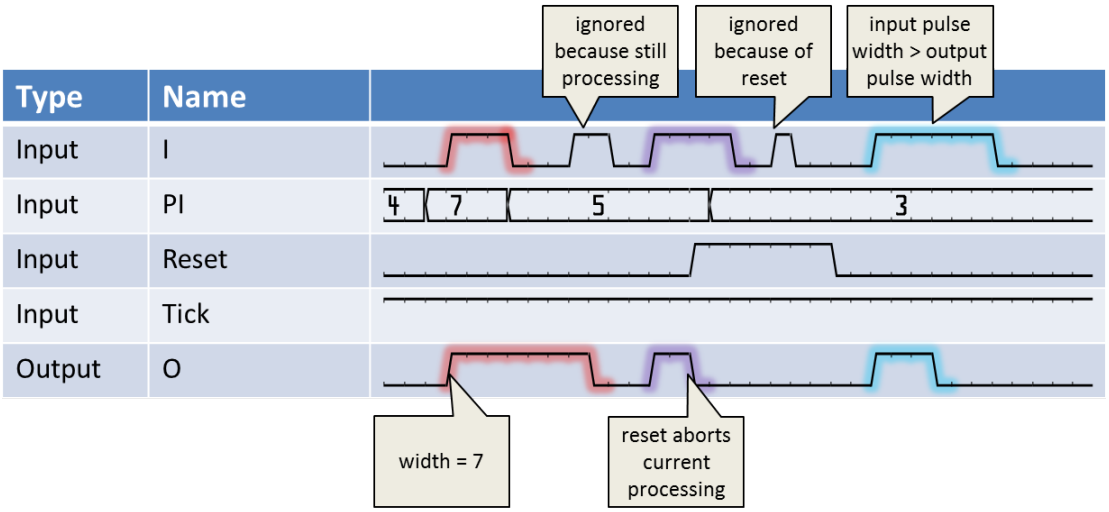
**Operator Library: Signal**

The operator generates a pulse for each rising edge at the input I. The pulse width is controlled by input PI.

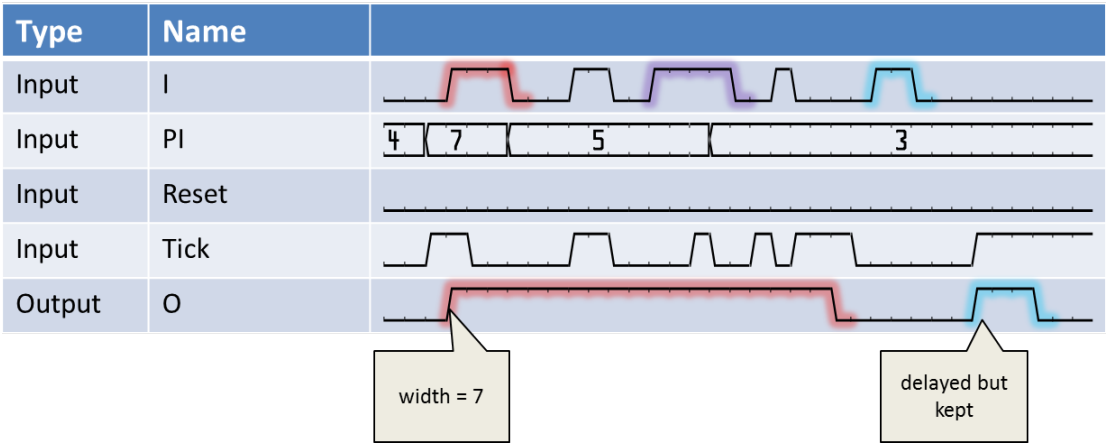
A rising edge at the input starts the pulse generation. The pulse width is equal to the last valid value at input link PI and is measured in ticks being high.

During a pulse generation no new rising edges at the operator input can be accepted. In this case, every new input rising edge will be ignored. Moreover, a change of the value at PI does not change the current pulse processing. PI is sampled at the occurrence of a rising edge at the input.

The operator can be reseted using input link Reset. While the reset input is high, no output pulses are processed. Any processing is aborted. The operator restarts operation when the reset input is low. The following waveform illustrates the operator's behavior.




The Tick input defines the time, the operator is processing data. It can be used like a prescaler. In most cases, the Tick input is not required. Tie it to operator VCC in this case. In the following figure, the influence of the Tick input is shown.




One special case when using ticks is that input pulses are sampled even if no tick is present. This is shown for the second input pulse of the waveform. This ensures that no input pulses can get lost.

This operator is excluded from the VisualApplets functional simulation.

### 31.34.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Links** | I, signal input<br>PI, control image data input<br>Tick, signal input<br>Reset, signal input |
| **Output Link** | O, signal output |

**Synchronous and Asynchronous Inputs**

- All signal inputs may be sourced by the same or different M-type operators through an arbitrary network of O-type operators. If they are sourced by the same M-type source, they will be automatically synchronized.
- Input link *PI* is asynchronous to the signal inputs.

### 31.34.2. Supported Link Format

| Link Parameter | Input Link I | Input Link PI | Input Link Tick |
|---|---|---|---|
| **Bit Width** | 1 | [1, 64] | 1 |
| **Arithmetic** | unsigned | unsigned | unsigned |
| **Parallelism** | 1 | 1 | 1 |
| **Kernel Columns** | 1 | 1 | 1 |
| **Kernel Rows** | 1 | 1 | 1 |
| **Img Protocol** | VALT_SIGNAL | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | VALT_SIGNAL |
| **Color Format** | VAF_GRAY | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE | FL_NONE |
| **Max. Img Width** | any | any | any |
| **Max. Img Height** | any | any | any |

| Link Parameter | Input Link Reset | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I |
| **Arithmetic** | unsigned | as I |
| **Parallelism** | 1 | as I |
| **Kernel Columns** | 1 | as I |
| **Kernel Rows** | 1 | as I |
| **Img Protocol** | VALT_SIGNAL | as I |
| **Color Format** | VAF_GRAY | as I |
| **Color Flavor** | FL_NONE | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

### 31.34.3. Parameters

None

### 31.34.4. Examples of Use

The use of operator WidthToSignal is shown in the following examples:

- Section 13.5, 'Functional Example for Specific Operators of Library Signal'

  Examples - Demonstration of how to use the operator

# 32. Library Synchronization




The library includes operators for image synchronizations such as removing, appending, splitting, inserting, ... of images, lines and pixels.

The following list summarizes all Operators of Library Synchronization

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | AppendImage | Appends images to a single image. | Version 1.2 |
|  | AppendImageDyn | Dynamically appending images controlled by the input *Append* (as opposed to the operator *AppendImage*, in which the amount of concatenated images is defined by a parameter). | Version 3.3 |
|  | AppendLine | Appends multiple image lines into a single line. | Version 1.2 |
|  | AppendLineDyn | Dynamically appending several image lines into a single line controlled by the input *Append* (as opposed to the operator *AppendLine*, in which the amount of concatenated lines is defined by a parameter). | Version 3.3 |
|  | CutImage | Dynamically cuts an image into a smaller one (truncate mode) or into several separate images (split mode), which is controlled by the input *Cut*. | Version 3.3 |
|  | CutLine | Dynamically cutting lines of an image into smaller ones (truncate mode) or into several separate lines (split mode), which is controlled by the input *Cut*. | Version 3.3 |
|  | CreateBlankImage | The operator generates a binary image of the specified width and height. | Version 1.3 |
|  | ExpandLine | Replaces all lines of the output image by the last line of the previous input image. | Version 1.2 |
|  | ExpandPixel | Copies the last pixel of a line or frame to all pixels in the next line or frame. | Version 1.2 |
|  | ImageValve | This operator is a valve which controls the data flow of the input link. | Version 1.3 |
|  | InsertImage | Multiplexes input images in sequential order. | Version 1.2 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | InsertLine | Multiplexes input lines in sequential order. | Version 1.2 |
|  | InsertPixel | Insert pixels into an image. | Version 1.2 |
|  | IsFirstPixel | Marks the first pixel value of a line (in mode=line) / of a frame (in mode=frame). | Version 3.0.5 |
|  | IsLastPixel | Marks the last pixel of a line (in line mode) / of a frame (in frame mode). Can also be used to mark empty lines (in line mode) or frames (in frame mode). | Version 3.0.5 |
|  | PixelReplicator | Replicates the pixels at the input link. | Version 3.0.1 |
|  | PixelToImage | Allows to synchronize the 0D pixel stream with an 0D, 1D, or 2D image. | Version 1.3 |
|  | RemoveImage | Completely remove images. | Version 1.1 |
|  | RemoveLine | Completely removes image lines. | Version 1.2 |
|  | RemovePixel | Removes specified pixels from images. | Version 1.2 |
|  | ReSyncToLine | Accepts the data at the end of the line and synchronize it with the current image line. | Version 1.2 |
|  | RxImageLink | Receives images from a TxImageLink operator in the design. | Version 3.0 |
|  | SourceSelector | The operator provides a switch between n M-Type sources. | Version 1.3 |
|  | SplitImage | Splits images into new images of specified image height. | Version 1.2 |
|  | SplitLine | Splits an image line into multiple lines of specified width. | Version 1.2 |

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | SYNC | Synchronizes image streams. | Version 1.1 |
|  | TxImageLink | Sends images to an RxImageLink operator any place in the design. | Version 3.0 |
|  | Overflow | Overflow management for non-stoppable data stream. | Version 3.3 |

Table 32.1. Operators of Library Synchronization

# 32.1. Operator AppendImage

**Operator Library: Synchronization**

The operator AppendImage concatenates input images into a larger output image. The number of images assembled at the output link O is defined by the parameter *AppendNumber*. For example, when *AppendNumber* is set to 2, the operator will assemble two successive images into a new larger one.

Note that the operator reduces the frame rate on its output by keeping the original bandwidth. The frame rate is reduced by factor *AppendNumber*.

The output link image protocol of the operator can also be set to VALT_LINE1D. In this case, the operator appends all incoming images into one image of unlimited height. This way, the operator can be used to convert a 2D image stream into a 1D line stream. If the 2D to 1D conversion is used in the VisualApplets simulation, images are not concatenated. In simulation, for every input image, one output image is generated.

When changing the *AppendNumber* value dynamically while acquisition is running, the operator guarantees image integrity. The operator keeps the old *AppendNumber* value until the output frame is finished. After the completion of the current output frame the operator will start using the new *AppendNumber* value for further merging.



**Exceeded Maximum Image Height**

Please note that it is possible to exceed the maximum image height defined in the output link with this operator. Always ensure that the concatenated input images do not exceed the parameterized maximum image height of the output link. Further operators might not work correctly if the image height is exceeded.

For example, 3 input images of height 1024 are concatenated. The output image height will be 3072. Thus, the maximum image height in the output link must be set to 3072 or higher to ensure a correct functionality of the VisualApplets operators.



**Converting 1D to 2D**

If you want to perform a 1D to 2D conversion, you can use operator SplitImage. See *documentation on operator SplitImage*.

## 32.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 32.1.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D, VALT_LINE1D |
| **Color Format** | any | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | any❷ |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The maximum output image height has to be greater or equal than parameter *AppendNumber* times the input image height. However, if the output link image protocol is set to VALT_LINE1D, the parameter *AppendNumber* is deactivated.

### 32.1.3. Parameters

| **AppendNumber** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 1 |
| **Range** | [1, Max. Image Height(output link)] |
| The number of input images which are concatenated into a single output image.<br><br>This parameter can only be changed if the output link image protocol is set to VALT_IMAGE2D. | |

| **LinesToSimulate** | |
|---|---|
| **Type** | static read/write parameter |
| **Default** | 1 |
| **Range** | [1, 2^32 - 1] |
| The number of lines in the simulated 1D output stream.<br><br>This parameter can only be used during simulation if the output protocol is set to VALT_IMAGE1D.<br><br>Background information: In VisualApplets simulation, there are always 2-dimensional images created. To be able to simulate lines, this parameter has been created (for simulation only). The parameter allows to specify the number of lines that are to be simulated. During simulation, these lines are put together to one 2-dimensional "image". This way, the individual lines can be simulated. All lines that come in after the number specified in LinesToSimulate has been reached are discarded. Exactly one 2-dimensional output image is generated for simulation. | |

### 32.1.4. Examples of Use

The use of operator AppendImage is shown in the following examples:

- Section 11.5, 'Functional Example for Multi Tap Camera Interface with Tap Geometry Sorting '

  Examples - Demonstration of how to use the operator

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

- Section 12.10.3, 'Image Composition Using Exposure Fusion'

  Examples - ExposureFusion

# 32.2. Operator AppendImageDyn

**Operator Library: Synchronization**

The operator *AppendImageDyn* dynamically concatenates input images into a larger output image. Whether images are appended is controlled by the input *Append*, which is synchronous to input *I*. This is the difference to the operator *AppendImage*, in which the amount of concatenated images is defined by a parameter. Whenever any of the pixels of the input *Append* has the value 1, the following image is appended to the current image.



**Do Not Exceed the Maximum Image Height**

By using this operator, it is possible to exceed the maximum image height as defined in the output link. Therefore, always ensure that the concatenated input images do not exceed the parameterized maximum image height of the output link. Operators that follow in the design might not work correctly if the image height is exceeded.

## 32.2.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I, data input<br>Append, control input |
| **Output Link** | O, data output |

## 32.2.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Append | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | as I | VALT_IMAGE2D |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | any❷ |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

❷ The maximum output image height has to be greater or equal than the input image height.

## 32.2.3. Parameters

None

### 32.2.4. Examples of Use

The use of operator AppendImageDyn is shown in the following examples:

- Section 13.2, 'Functional Example for Specific Operators of Library Synchronization: Dynamic Append and Cut'

  Examples - Demonstration of how to use the operator

# 32.3. Operator AppendLine

**Operator Library: Synchronization**

The operator *AppendLine* concatenates image lines from the input image at the link I into a larger line in the output image at the link O. The number of lines merged into a new line is defined by the parameter *AppendNumber*. For example, when *AppendNumber* is set to 2, the operator will concatenate 2 successive lines from the input image into a new larger line in the output image. When the input image height is not divisible by the *AppendNumber* parameter value, the last line in the output image will be shorter and will contain the remaining last lines of the input image concatenated together.

Note that the operator does not change the frame rate nor does it change the bandwidth.

The output link image protocol of the operator can also be set to VALT_PIXEL0D, i.e. the operator appends all incoming lines into a line of unlimited width. Thus, the operator can be used to convert a 2D image stream or 1D line stream into a 0D pixel stream.

By changing the *AppendNumber* value dynamically while acquisition is running, the operator guarantees line integrity. The operator keeps the old *AppendNumber* value until the output line is finished. After the completion of the current output line, the operator will start using the new *AppendNumber* value for further merging.



**Prevent Exceeding the Maximum Image Dimensions**

By using this operator, it is possible to exceed the maximum image width and height defined in the output link. Therefore, always ensure that the concatenated input lines do not exceed the parameterized maximum image width of the output link. Also make sure that the parameterized maximum image height of the output link is large enough. Subsequent operators may not work correctly, if the image width or height exceeds the limits specified by the link properties.

For example, 3 input lines of width 1024 are concatenated. The output image width will be 3072. This width has to be specified in the output link to ensure a correct functionality of the VisualApplets operators.

## 32.3.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 32.3.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | any❷ |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Max. Img Height** | any | any |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The maximum image width has to be greater than or equal to parameter *AppendNumber* times the input image width.

### 32.3.3. Parameters

| **AppendNumber** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 1 |
| **Range** | Input IMAGE2D: [1, Input Maximum Image Height]; Input LINE1D: [1, Output Maximum Image Width / Parallelism] |
| The number of input lines which are concatenated into a single output line.<br><br>This parameter can only be changed, if the output link image protocol is not set to VALT_PIXEL0D. | |

| **LinesToSimulate** | |
|---|---|
| **Type** | static read/write parameter |
| **Default** | 1 |
| **Range** | [1, 2^32 - 1] |
| The number of lines in the simulated 1D output stream.<br><br>This parameter can only be used during simulation, if the input image protocol is set to VALT_IMAGE2D and the output protocol is set to VALT_IMAGE1D.<br><br>Exactly one output image is generated. Remaining input lines are discarded. | |

| **PixelsToSimulate** | |
|---|---|
| **Type** | static read/write parameter |
| **Default** | 1 |
| **Range** | [1, 2^32 - 1] |
| The number of input pixels in the simulated 0D output stream.<br><br>This parameter can only be used during simulation, if the input image protocol is set to VALT_IMAGE2D or VALT_IMAGE1D and the output protocol is set to VALT_IMAGE0D.<br><br>Exactly one output image is generated. Remaining input pixels are discarded. | |

### 32.3.4. Examples of Use

The use of operator AppendLine is shown in the following examples:

- Section 9.3.1.4, 'Stitching of Two Cameras'

  Tutorial - Use of the operator for stitching the images of two cameras.

- Section 12.9.9, 'Tap Geometry Sorting'

  Examples - Scaling A Line Scan Image

# 32.4. Operator AppendLineDyn

**Operator Library: Synchronization**

The operator *AppendLineDyn* dynamically concatenates image lines from the input image at the link *I* into a larger line in the output image at the link *O*. Whether lines shall be appended is controlled by the input *Append*, which is synchronous to input *I*. This is the difference to the operator *AppendLine*, in which the amount of concatenated lines is defined by a parameter. Whenever any of the pixels of a line at the input *Append* has the value 1, the following line is appended to the current line.

The operator does not change the frame rate nor does it change the bandwidth.

Using this operator may typically result in images with different line widths depending on the pattern on input *Append*.



**Do Not Exceed the Maximum Image Dimensions**

By using this operator, it is possible to exceed the maximum image width and height as defined in the output link. Therefore, always ensure that the concatenated input lines do not exceed the parameterized maximum image width of the output link. Also make sure that the parameterized maximum image height of the output link is large enough. Operators that follow in the design may not work correctly, if the image width or height exceeds the limits specified by the link properties.

For example, 3 input lines with the width 1024 are concatenated. The output image width will be 3072. Thus, the maximum image width in the output link must be set to 3072 or higher to ensure a correct functionality of the VisualApplets operators.

## 32.4.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I, data input<br>Append, control input |
| **Output Link** | O, data output |

## 32.4.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Append | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I | as I |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | as I | any❷ |
| **Max. Img Height** | any | as I | any |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]

- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

❷ The maximum image width has to be greater or equal than the input image width.

### 32.4.3. Parameters

None

### 32.4.4. Examples of Use

The use of operator AppendLineDyn is shown in the following examples:

- Section 13.2, 'Functional Example for Specific Operators of Library Synchronization: Dynamic Append and Cut'

  Examples - Demonstration of how to use the operator

# 32.5. Operator CutImage

**Operator Library: Synchronization**

The operator *CutImage* dynamically cuts the input image into a number of smaller images. Cutting is controlled by the input *Cut*, which is synchronous to input *I*. The operator supports two cutting modes which are set by the parameter *Mode*:

- If *Mode* is set to *Split* (default), the incoming image is split into several pieces. Whenever any of the pixels of the current line at input *Cut* has the value 1, the ongoing output image is terminated at the end of that line and a new image is started with the beginning of the next incoming line. Any value of 1 in the last line of an image at input *Cut* has no effect.
- If *Mode* is set to *Truncate* the incoming image is cut to a single image which may have less lines than the incoming image. Whenever any of the pixels of the current line at input *Cut* has the value 1, the ongoing output image is terminated at the end of that line. No further image data is forwarded to the output *O* until the end of the current image.



**Increased Frame Rate**

In mode *Split*, the operator may increase the frame rate by keeping the original bandwidth. In mode *Truncate*, the operator doesn't change the frame rate but may reduce the mean bandwidth.

You can use the operator *CutImage* to convert a 1D image into a 2D image. The conversion is automatically performed, if a link using the image protocol VALT_LINE1D is connected to the operator's input. In this case the operator is always in *Split* mode and the parameter *Mode* is disabled. For example, the 1D input of a line scan camera can be split into images of varying height depending on the control input, effectively creating a 2D output stream.

The operator supports images with variable line length.

## 32.5.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I, data input<br>Cut, control input |
| **Output Link** | O, data output |

## 32.5.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Cut | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I | VALT_IMAGE2D |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | as I | as I |
| **Max. Img Height** | any | as I | as I |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]
- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

### 32.5.3. Parameters

| **Mode** | |
|---|---|
| **Type** | static read/write parameter |
| **Default** | Split |
| **Range** | {Split,Truncate} |

Operation mode.

- If *Mode* is set to *Split* (default), the incoming image is split into several pieces. Whenever any of the pixels of the current line at input *Cut* has the value 1, the ongoing output image is terminated at the end of that line and a new image is started with the beginning of the next incoming line. Any value of 1 in the last line of an image at input *Cut* has no effect.
- If *Mode* is set to *Truncate* the incoming image is cut to a single image which may have less lines than the incoming image. Whenever any of the pixels of the current line at input *Cut* has the value 1, the ongoing output image is terminated at the end of that line. No further image data is forwarded to the output *O* until the end of the current image.

### 32.5.4. Examples of Use

The use of operator CutImage is shown in the following examples:

- Section 13.2, 'Functional Example for Specific Operators of Library Synchronization: Dynamic Append and Cut'

  Examples - Demonstration of how to use the operator

# 32.6. Operator CutLine

**Operator Library: Synchronization**

The operator *CutLine* dynamically cuts lines of an input image into a number of shorter lines. Cutting is controlled by the input *Cut*, which is synchronous to input *I*. Cutting is done at the granularity given by parallelism P. Therefore, no dummy pixels are generated. When a pixel value at input *Cut* is 1, the corresponding pixel from input *I* is still contained in the output of the current line. The line is terminated after the next possible pixel position so the granularity condition is met. The operator supports two cutting modes which are set by the parameter *Mode*:

- If *Mode* is set to *Split* (default), the incoming lines are split into several pieces. Whenever a pixel of the current line at input *Cut* has the value 1, the ongoing output line is terminated at the granularity given by parallelism P. A new line is then started with the beginning of the next incoming pixels. Any value of 1 in the last parallelism P pixels of a line has no effect.
- If *Mode* is set to *Truncate*, the incoming line is cut to a single line which may be shorter than the incoming line. Whenever a pixel of the current line at input *Cut* has the value 1, the ongoing output line is terminated at the granularity given by parallelism P. No further pixels are forwarded to the output *O* until the end of the current line.

The operator might generate output images with different line lengths. Not all VisualApplets operators can process images using varying line lengths.

You can also use the *CutLine* operator to convert a 0D image into a 1D image. The conversion is automatically performed, if a link using the image protocol VALT_PIXEL0D is connected to the operator's input. In this case the operator is always in *Split* mode and the parameter *Mode* is disabled.

## 32.6.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, data input<br>Cut, control input |
| **Output Link** | O, data output |

## 32.6.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Cut | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | as I | auto❷ |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | as I | any❸ |
| **Max. Img Height** | any | as I | any❹ |

❶ The range of the input bit width is:

- For unsigned inputs: [1, 64]

- For signed inputs: [2, 64]
- For unsigned color inputs: [3, 63]
- For signed color inputs: [6, 63].

❷ If VALT_IMAGE2D or VALT_LINE1D is used, the output image protocol is the same as the input image protocol. If VALT_PIXEL0D image protocol is used at the input, the operator converts the image into the VALT_LINE1D image protocol.

❸ The output maximum image width has to be less than or equal to the input maximum image width.

❹ The output maximum image height has to be greater than or equal to the input image height.

## 32.6.3. Parameters

| **Mode** | |
|---|---|
| **Type** | static read/write parameter |
| **Default** | Split |
| **Range** | {Split,Truncate} |

Operation mode.

- If *Mode* is set to *Split* (default), the incoming lines are split into several pieces. Whenever a pixel of the current line at input *Cut* has the value 1, the ongoing output line is terminated at the granularity given by parallelism P and a new line is started with the beginning of the next incoming pixels. Any value of 1 in the last parallelism P pixels of a line has no effect.
- If *Mode* is set to *Truncate*, the incoming line is cut to a single line which may be shorter than the incoming line. Whenever a pixel of the current line at input *Cut* has the value 1, the ongoing output line is terminated at the granularity given by parallelism P. No further pixels are forwarded to the output *O* until the end of the current line.

## 32.6.4. Examples of Use

The use of operator CutLine is shown in the following examples:

- Section 13.2, 'Functional Example for Specific Operators of Library Synchronization: Dynamic Append and Cut'

  Examples - Demonstration of how to use the operator

# 32.7. Operator CreateBlankImage

**Operator Library: Synchronization**

This operator generates a blank binary image of the specified width and height. The operator does not require any input links. The output image width and height is specified using parameters *ImageWidth* and *ImageHeight*.

The operator will output the blank images at the maximum possible speed. Thus the output speed is only limited by the output link parallelism and the following operators. The VisualApplets flow control will limit the bandwidth if further operators cannot process the data rates produces by CreateBlankImage.

Often, this operator is used to test the functionality of an implementation without the need to connect a camera. Using operators such as *Coordinate_x*, *Coordinate_y* and *ImageNumber*, the operator can be used to generate test patterns. In conjunction with operator *ImageValve* the frame, line or pixel rate can be controlled.

The image dimensions of the generated images can only be changed when the acquisition is not running i.e. stopped.

In case of 1D application the *ImageHeight* parameter is ignored. In case of 0D application both *ImageWidth* and *ImageHeight* are ignored.

The operator can easily be used with the VisualApplets simulation. The operator generates simulation images itself. It is working as a simulation source. Therefore, it is not required to add a simulation source probe to the output link of the operator.



**Behavior during simulation**

The operator outputs at least one data set per simulation step. Depending on the current link configuration at the output port this data set is

- a series of pixels in 0D mode. The amount of pixels is controlled via parameter *PixelsToSimulate*
- a series of lines in 1D mode. The amount of lines is controlled via parameter *LinesToSimulate*
- a series of frames in 2D mode. The amount of frames is controlled via parameter *FramesToSimulate*

The default value of those parameters is 1. Depending on the current link configuration at the output port only the relevant parameter is editable and the others are inactive. I.e., for 0D only *PixelsToSimulate* is editable while for 1D the parameters *LinesToSimulate* and *ImageWidth* are editable.

The operator guarantees image integrity while parameters *ImageHeight* or *ImageWidth* are changed during acquisition. After the completion of the current frame output the operator will start using the new parameter values.

## 32.7.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Output Link** | O, data output |

## 32.7.2. Supported Link Format

| Link Parameter | Output Link O |
|---|---|
| **Bit Width** | 1 |

| Link Parameter | Output Link O |
|---|---|
| **Arithmetic** | unsigned |
| **Parallelism** | any |
| **Kernel Columns** | 1 |
| **Kernel Rows** | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} |
| **Color Format** | VAF_GRAY |
| **Color Flavor** | FL_NONE |
| **Max. Img Width** | any |
| **Max. Img Height** | any |

### 32.7.3. Parameters

| ImageWidth | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [0, Maximum Output Image Width], step size = output parallelism |
| Specifies the width of the generated images. This parameter is enabled only if the operator's output image protocol is set to VALT_IMAGE2D or VALT_LINE1D. | |

| ImageHeight | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1024 |
| **Range** | [1, Maximum Output Image Height] |
| Specifies the height of the generated images. This parameter is enabled only if the operator's output image protocol is set to VALT_IMAGE2D. | |

| FramesToSimulate | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | [1, 2^32 - 1] |
| If the output link image protocol is set to VALT_IMAGE2D, this parameter specifies the number of frames which are generated for a single simulation step. See Section 3.10, 'Simulation' for more information on 2D simulations. This parameter can only be changed, if the output image protocol is set to VALT_IMAGE2D. | |

| LinesToSimulate | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | [1, 2^32 - 1] |
| If the output link image protocol is set to VALT_LINE1D, it is required to specify the number of lines for simulation. This parameter is used to specify the simulation image height. See Section 3.10, 'Simulation' for more information on 1D simulations. This parameter can only be changed if the output image protocol is set to VALT_LINE1D. | |

| PixelsToSimulate | |
|---|---|
| **Type** | static write parameter |
| **Default** | 1 |
| **Range** | [1, 2^32 - 1], step size = output parallelism |

| PixelsToSimulate |
| --- |
| If the output link image protocol is set to VALT_PIXEL0D, it is required to specify the size of the pixel stream for simulation. This parameter is used to specify the simulation pixel stream width. See Section 3.10, 'Simulation' for more information on 0D simulations. This parameter can only be changed if the output image protocol is set to VALT_LINE0D. |

### 32.7.4. Examples of Use

The use of operator CreateBlankImage is shown in the following examples:

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.6.2, 'Image Dimension Test'

  Example - The image dimension is measured and can be used to analyze the design flow.

- Section 12.6.4, 'Manual Image Injection'

  Example - For debugging purposes images can be inserted manually.

- Section 12.6.5, 'Image Monitoring'

  Example - For debugging purposes image transfer states on links can be investigated.

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

# 32.8. Operator ExpandLine

**Operator Library: Synchronization**

The operator ExpandLine stores the last image line of each input image. This stored image line will used to replace all image lines of the next image at the output. In other words, the operator copies the last image line to all lines of the next image.

Mind, that the behavior of this operator will only affect the following frame and not the current frame. Therefore, the output data of the first frame after acquisition start has to be be specified using parameter *Init*.

If the width of a successive frame is less than the previous frame, the operator will keep the image dimension and only output the first part of the stored line. If the width of a successive frame is greater than the previous frame, the operator will fill the missing pixels with random memory content.

**Operator Restrictions**

- Empty images are not supported.

  Images with varying line lengths are not supported.

  Input image dimension must be equal to the maximum image dimensions set in the link.

## 32.8.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 32.8.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | VALT_IMAGE2D | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 32.8.3. Parameters

| Init | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 2^InputBitWidth - 1] |

| Init |
| --- |
| The parameter specifies the initialization value of the pixels of the first output frame. The parameter will only accept unsigned values. For signed initialization values enter its unsigned representation. |

## 32.8.4. Examples of Use

The use of operator ExpandLine is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

  Examples - Demonstration of how to use the operator

# 32.9. Operator ExpandPixel

**Operator Library: Synchronization**

The operator *ExpandPixel* copies the last pixel of a line or frame to all pixels in the next line or frame. The operator is used in two modes:

- **Line Sync** mode:

  In this mode, for each line the last pixel value in a line is stored and kept constant for the next line.

  To use this mode, set the parameter *AutoSync* to *EoL*.

- **Frame Sync** mode:

  In this mode for each frame the last pixel in the image is stored and kept constant during the next frame.

  To use this mode, set the parameter *AutoSync* to *EoF* .

Mind that the behavior of this operator will only affect the following line or frame and not the current line or frame. For this reason the output for the first line or frame has to be specified via the parameter *Init*.

When using the **Line Sync** mode with a 2D protocol, then the last pixel value of the last line of a frame is used for the output of pixels of the first line of the following frame. So the value of parameter *Init* is only used for the very first frame.

**Operator Restrictions**

- Empty images are not supported.

  Images with varying line lengths are not supported.

## 32.9.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 32.9.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

### 32.9.3. Parameters

| Init | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 2^InputBitWidth - 1] |
| The parameter specifies the initialization value of of the pixels of the first image lines if *AutoSync* is set to EoL or the parameter specifies the pixels of the first output frame if *AutoSync* is set to Eof. The parameter will only accept unsigned values. For signed initialization values enter its unsigned representation. | |

| AutoSync | |
|---|---|
| **Type** | static parameter |
| **Default** | EoL |
| **Range** | {EoL, EoF} |
| Specifies the synchronization mode of the operator. Use mode EoL to copy the last pixel of each line to all pixels of the following line. Use mode EoF to copy the last pixel of each frame to all pixels of the following frame. If the image protocol at the output is set to VALT_LINE1D, the parameter can only be set to EoL. | |

### 32.9.4. Examples of Use

The use of operator ExpandPixel is shown in the following examples:

- Section 12.2.2, 'Auto Threshold Mean'

  Determines the mean value of an image and used the value as threshold value for the next image processed.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

# 32.10. Operator ImageValve

**Operator Library: Synchronization**

This operator can be used as a valve to control the data flow of the input link. The signal input *OpenValve* is used to control the valve. Hence, the operator allows to influence the data flow of pixels, lines and frames between I and O. Limiting the maximum bandwidth is the main purpose of this operator. Broadly, this operator is used to block and to control the speed of image processing. For example, this operator can be used in conjunction with *CreateBlankImage* to implement an image generator with controllable output frame rate.

If the valve is closed, the VisualApplets flow control will block all incoming data. The input pipeline will then be blocked. Ensure that a sufficiently sized buffer is used if operators are used that cannot stop their processing. An example are free running cameras connected to the frame grabber. If the output of the operator is blocked by any successive operator, the operator will forward this block to the input. In this case the opening of the valve has no influence. Do not mix up the valve with a gate. A gate operator discards the input data while it is closed.

The valve is controlled by the signal input *OpenValve* and the operator's parameters. If *OpenValve* is set to one, the operator allows the pass of *SequenceLength* pixels, lines or frames depending on the parameter settings of *Mode*.

If *OpenValve* is set to a constant one the operator allows an unblocked pass of the image data from input link I to output link O, i.e. the valve is open all the time.

If a second pulse is set to OpenValve while the operator's valve is still open and a sequence is processed, the pulse is ignored. If the sequence length is changed using parameter *SequenceLength* while currently a sequence is processed, the operator first finishess the previous sequence and will use the new value afterwards.

The high level simulation in VisualApplets will pass all input images directly to the output as no timing is simulated. Any values at input link *OpenValve* are ignored.

## 32.10.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, data input<br>OpenValve, control signal input |
| **Output Link** | O, data output |

## 32.10.2. Supported Link Format

| Link Parameter | Input Link I | Input Link OpenValve | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | 1 | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | VALT_SIGNAL | as I |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | any | as I |
| **Max. Img Height** | any | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

**Synchronous and Asynchronous Inputs**

- All inputs are asynchronous to each other.

## 32.10.3. Parameters

| **Mode** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | FRAME |
| **Range** | {PIXEL, LINE, FRAME} |
| Set the mode of the operator. Depending on this parameter the valve opens for *SequenceLength* pixels, lines or frames.<br><br>The availability of modes LINE and FRAME depends on the image protocol of input link I. | |

| **SequenceLengthBits** | |
|---|---|
| **Type** | static parameter |
| **Default** | 10 |
| **Range** | [1, 64] |
| Set the number of bits to represent the sequence length.<br><br>This parameter is activated only if parameter *SequenceLength* is set to dynamic. | |

| **SequenceLength** | |
|---|---|
| **Type** | dynamic/static read/write parameter |
| **Default** | 1 |
| **Range** | dynamic: [1, 2^SequenceLengthBits - 1], static: [1, 2^64 - 1] |
| Set the length of the sequence. The operator will output the given number of pixels, lines or frames depending on parameter *Mode* if the operator is in open state.<br><br>If Mode is set to PIXEL, the value has to be divisible by the parallelism. | |

## 32.10.4. Examples of Use

The use of operator ImageValve is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.6.7, 'Image Flow Control'

  Example - For debugging purposes of the designs internal data flow control in hardware and a possible compensation.

# 32.11. Operator InsertImage

**Operator Library: Synchronization**

The operator InsertImage multiplexes a number of n input links I[0] .. I[n-1] into the output link O. Thus the operator outputs the input images of all inputs in sequential order at O.

The operator forwards the input images to the output one after the other. First, input link I[0] is processed, next link I[1]. After link I[n-1] the operator starts with the next image at I[0] again etc. The operator waits until an image at a currently selected input is present. Therefore it is not possible to skip inputs. There is no "first come first processed" strategy.

As explained, the operator only forwards a specific input at a certain time. While one input is processed all other inputs are blocked. If an image is present at a blocked input, ensure that it can be buffered until it's processed. Moreover, users have to ensure that multiple inputs are not sourced by the same image source as this will cause a deadlock. Check tutorial Section 9.3.1.3, 'Multiplex the Images of Two Cameras' for more information on correct inserting.

For every input I[k], the operator provides a controlling input *Ins[k]*. These inputs are used to control whether the data of an image has to be used and forwarded to the output or should be discarded. The *Ins[k]* inputs accept binary data. If a one is provided for the very first pixel of a respective image, the image will be forwarded to the output. Whereas if value zero is provided for the very first pixel of a respective image, the image will be processed but discarded.

The I[k] and *Ins[k]* pairs are synchronous i.e. they have to be sources by the same M-type source though an arbitrary network of O-type operators. See Section 3.6.4, 'M-type Operators with Multiple Inputs' for more information. Therefore, no SYNC operator is required for these pairs (different in VA versions < VA2).

The output frame rate of the operator is n times higher than the input frame rate. However, the output parallelism is not increased. Therefore, the output bandwidth is equal to the input bandwidth while more data might be required to be transferred. It might be necessary to increase the parallelism of the inputs before the operator is processing the data.

All images at I[k] can be of different height and width. Control inputs *Ins[k]* are ignored for empty input images. These images will not be forwarded to the output i.e. will be discarded.

## 32.11.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I[0], data input<br>I[k], data input<br>Ins[k], control input |
| **Output Link** | O, data output |

## 32.11.2. Supported Link Format

| Link Parameter | Input Link I[0] | Input Link I[k] |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I[0] |
| **Arithmetic** | {unsigned, signed} | as I[0] |
| **Parallelism** | any | as I[0] |
| **Kernel Columns** | any | as I[0] |
| **Kernel Rows** | any | as I[0] |
| **Img Protocol** | VALT_IMAGE2D | as I[0] |
| **Color Format** | any | as I[0] |
| **Color Flavor** | any | as I[0] |

| Link Parameter | Input Link I[0] | Input Link I[k] |
|---|---|---|
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | any |

| Link Parameter | Input Link Ins[k] | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I[0] |
| **Arithmetic** | unsigned | as I[0] |
| **Parallelism** | as I[0] | as I[0] |
| **Kernel Columns** | 1 | as I[0] |
| **Kernel Rows** | 1 | as I[0] |
| **Img Protocol** | as I[0] | as I[0] |
| **Color Format** | VAF_GRAY | as I[0] |
| **Color Flavor** | FL_NONE | as I[0] |
| **Max. Img Width** | as I[k] | auto❷ |
| **Max. Img Height** | as I[k] | auto❸ |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The maximum output image width is the maximum of all I[k] input image widths.
❸ The maximum output image height is the maximum of all I[k] input image heights.

**Synchronous and Asynchronous Inputs**

- Synchronous Groups: I[k] and Ins[k]
- All groups are asynchronous to each other.

## 32.11.3. Parameters

None

## 32.11.4. Examples of Use

The use of operator InsertImage is shown in the following examples:

- Section 9.3.1.3, 'Multiplex the Images of Two Cameras'

  Tutorial - Use of the operator for multiplexing the images of two cameras.

- Section 11.5, 'Functional Example for Multi Tap Camera Interface with Tap Geometry Sorting '

  Examples - Demonstration of how to use the operator

- Section 12.4.2.2, 'Color Plane Separation Option 2 - Three Buffers, One DMA'

  Sequential output of the color planes using three image buffers and one DMA operator.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

- Section 12.12.2, 'Depth From Focus Using Loops'

  Examples - Depth From Focus using Loops

# 32.12. Operator InsertLine

**Operator Library: Synchronization**

The operator InsertLine multiplexes the lines of N input links I[0] .. I[n-1] into the output link O. Thus the operator outputs the input image lines of all inputs in sequential order at O.

The operator forwards the input image lines to the output one after the other. With every new frame the operator starts with input I[0]. First, the first line of input I[0] is processed and forwarded to the output. Next, the first line of input I[1] is processed. After the first line of I[n-1] has been processed, the operator continuous with the second line of I[0] etc. The operator waits until the required line at the currently selected input is present. Therefore it is not possible to skip inputs. There is no "first come first processed" strategy.

As explained, the operator only forwards a specific input at a certain time. While one input is processed all other inputs are blocks. If an image line is present at a blocked input, ensure that it can be buffered until it's processed. Moreover, users have to ensure that multiple inputs are not sourced by the same image source as this will cause a deadlock. Check tutorial Section 9.3.1.3, 'Multiplex the Images of Two Cameras' for more information on correct inserting. The example is similar for the InserLine operator.

For every input I[k], the operator provides a controlling input *Ins[k]*. These inputs are used to control whether the data of an image line has to be used and forwarded to the output or should be discarded. The *Ins[k]* inputs accept binary data. If a one is provided for the very first pixel of a respective image line, the line will be forwarded to the output. Whereas if value zero is provided for the very first pixel of a respective image line, the line will be processed but discarded.

The I[k] and *Ins[k]* pairs are synchronous i.e. they have to be sources by the same M-type source though an arbitrary network of O-type operators. See Section 3.6.4, 'M-type Operators with Multiple Inputs' for more information. Therefore, no SYNC operator is required for these pairs (different in VA versions < VA2).

The output image size of the operator is n times higher than the input image sizes. However, the output parallelism is not increased. Therefore, the output bandwidth is equal to the input bandwidth while more data might be required to be transferred. It might be necessary to increase the parallelism of the inputs before the operator is processing the data.

All images at I[k] can be of different width. However, the height of all images has to be equal. Control inputs *Ins[k]* are ignored for empty input lines. These lines will not be forwarded to the output i.e. will be discarded.

## 32.12.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I[0], data input<br>I[k], data input<br>Ins[k], control input |
| **Output Link** | O, data output |

## 32.12.2. Supported Link Format

| Link Parameter | Input Link I[0] | Input Link I[k] |
|---|---|---|
| **Bit Width** | [1,64]❶ | as I[0] |
| **Arithmetic** | {unsigned, signed} | as I[0] |
| **Parallelism** | any | as I[0] |
| **Kernel Columns** | any | as I[0] |
| **Kernel Rows** | any | as I[0] |

| Link Parameter | Input Link I[0] | Input Link I[k] |
|---|---|---|
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as I[0] |
| **Color Format** | any | as I[0] |
| **Color Flavor** | any | as I[0] |
| **Max. Img Width** | any | any |
| **Max. Img Height** | any | as I[0] |

| Link Parameter | Input Link Ins[k] | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I[0] |
| **Arithmetic** | unsigned | as I[0] |
| **Parallelism** | as I[0] | as I[0] |
| **Kernel Columns** | 1 | as I[0] |
| **Kernel Rows** | 1 | as I[0] |
| **Img Protocol** | as I[0] | as I[0] |
| **Color Format** | VAF_GRAY | as I[0] |
| **Color Flavor** | FL_NONE | as I[0] |
| **Max. Img Width** | as I[k] | auto❷ |
| **Max. Img Height** | as I[k] | N * I[0]❸ |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The maximum output image width is the maximum of all input image widths.
❸ As of VisualApplets version 3.3.0, and only if in the **Simulation Build Setttings** the option **Simulate 1D Line By Line** is selected: If the *Image Protocol* is set to *VALT_LINE1D*, the maximum output image height is forced to *1*.

**Synchronous and Asynchronous Inputs**

- Synchronous Groups: I[k] and Ins[k]
- All groups are asynchronous to each other.

## 32.12.3. Parameters

None

## 32.12.4. Examples of Use

The use of operator InsertLine is shown in the following examples:

- Section 3.6.6, 'Timing Synchronization'

  Timing Synchronization - Using M-type Operators with synchronous input groups and avoiding deadlocks.

- Section 9.3.1.4, 'Stitching of Two Cameras'

  Tutorial - Use of the operator for stitching the images of two cameras.

- Section 11.5, 'Functional Example for Multi Tap Camera Interface with Tap Geometry Sorting '

  Examples - Demonstration of how to use the operator

- Section 12.4.2.4, 'Color Plane Separation Option 4 - Sequential with Operator ImageBufferMultiRoI and a pre-sort of the Color Planes'

Sequential DMA output of the color planes. The color separations is performed using operator ImageBufferMultiROI. An additional pre-sorting optimizes the bandwdith and resources.

- Section 12.9.9, 'Tap Geometry Sorting'

  Examples - Scaling A Line Scan Image

# 32.13. Operator InsertPixel

**Operator Library: Synchronization**

Library: Synchronization

The operator InsertPixel inserts arbitrary pixel from the image on input link I1 into the image on I0 and outputs the combined image at O.

The insertion is controlled using the binary input link Ins. If Ins = 1 for a specific pixel, the value at I1 is inserted into the image at I0. Thus the output line length is increased by the number of inserted pixel. For example if *Ins* == 1 for all pixel, the line length at O be doubled compared to the input line length. All output pixels at even pixel coordinates are originated from I0 and all odd pixel are from I1. If *Ins* == 0 for all pixels, the output image at O is equal to the image at I0.

All input links have to be synchronous i.e. they have to be sourced by the same M-type operator through an arbitrary network of O type operators.

The operator doubles the parallelism to avoid bandwidth limitations. Please note: It is possible that the number of inserted pixels is not a multiple of the parallelism. In this case, the operator has to add dummy pixels at the end of the line. The value of these dummy pixels is undefined. During VA simulation dummy pixels are set to zero for better visibility.

## 32.13.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I0, data input<br>I1, data input<br>Ins, control input |
| **Output Link** | O, data output |

## 32.13.2. Supported Link Format

| Link Parameter | Input Link I0 | Input Link I1 |
|---|---|---|
| **Bit Width** | [1, 64] | as I0 |
| **Arithmetic** | {unsigned, signed} | as I0 |
| **Parallelism** | any | as I0 |
| **Kernel Columns** | any | as I0 |
| **Kernel Rows** | any | as I0 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I0 |
| **Color Format** | any | as I0 |
| **Color Flavor** | any | as I0 |
| **Max. Img Width** | any | as I0 |
| **Max. Img Height** | any | as I0 |

| Link Parameter | Input Link Ins | Output Link O |
|---|---|---|
| **Bit Width** | 1 | as I0 |
| **Arithmetic** | unsigned | as I0 |
| **Parallelism** | as I0 | 2 * I0 |
| **Kernel Columns** | 1 | as I0 |
| **Kernel Rows** | 1 | as I0 |

| Link Parameter | Input Link Ins | Output Link O |
|---|---|---|
| **Img Protocol** | as I0 | as I0 |
| **Color Format** | VAF_GRAY | as I0 |
| **Color Flavor** | FL_NONE | as I0 |
| **Max. Img Width** | as I0 | 2 * I0❷ |
| **Max. Img Height** | as I0 | as I0 |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The output image width must not exceed 2^31 - 1.

**Synchronous and Asynchronous Inputs**

- All inputs are synchronous to each other i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

## 32.13.3. Parameters

None

## 32.13.4. Examples of Use

The use of operator InsertPixel is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

  Examples - Demonstration of how to use the operator

# 32.14. Operator IsFirstPixel

**Operator Library: Synchronization**

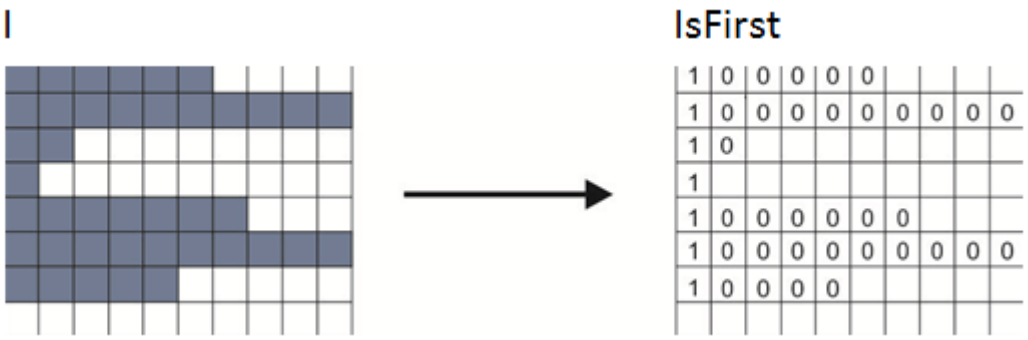
Operator *IsFirstPixel* marks the first pixel in a line (in line mode) or in a frame (in frame mode). The operator outputs a 1 on its output port *IsFirstO* for each first pixel of a line/frame.

Empty line: If the operator receives an empty line, it also outputs an empty line. *IsFirstO* is NOT set to 1 in this case.

Empty frame: If the operator receives an empty frame, it also outputs an empty frame. *IsFirstO* is NOT set to 1 in this case.

Operator *IsFirstPixel* is an O-type operator. The pixel values received on the input port I are not interpreted.

Example (parameter *Mode* = Line):




**Parallelism > 1**

If parallelism > 1, only the first pixel in the parallelism is marked with a 1 on output port *IsFirstO*. All other parallelism components are 0.

Example: If Parallelism = 4, *IsFirstO* is 0x1 at the first pixel, and in all other cases 0.

## 32.14.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | O |
| **Input Link** | I, data input |
| **Output Link** | IsFirstO, output of 1 for first pixel in line/frame and 0 for all other pixels |

## 32.14.2. Supported Link Format

| Link Parameter | Input Link I | Output Link IsFirstO |
|---|---|---|
| **Bit Width** | [1, 63] | 1 |
| **Arithmetic** | {unsigned, signed} | unsigned |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | 1 |
| **Kernel Rows** | any | 1 |
| **Img Protocol** | {VALT_PIXEL0D, VALT_LINE1D, VALT_IMAGE2D} | as I |

| Link Parameter | Input Link I | Output Link IsFirstO |
|---|---|---|
| **Color Format** | {VAF_COLOR, VAF_GRAY} | VAF_GRAY |
| **Color Flavor** | any | FL_NONE |
| **Max. Img Width** | any❶ | as I |
| **Max. Img Height** | any | as I |

❶ The maximum image width must be divisible by the parallelism.

## 32.14.3. Parameters

| Mode | |
|---|---|
| **Type** | static or dynamic write parameter |
| **Default** | Frame |
| **Range** | {Line, Frame} |

If set to "Line", the operator marks the first pixel in a line. The operator outputs a 1 on its output port IsFirstO for each first pixel of a line.

If set to "Frame", the operator marks the first pixel in a frame. The operator outputs a 1 on its output port IsFirstO for each first pixel of a frame.

This parameter you can set to static or dynamic. When you use the parameter as a dynamic parameter: A shadow register is implemented, and the change is taken over between

- two images (when I is VALT_IMAGE2D),
- two lines (when I is I is VALT_ LINE1D),
- directly (when I is I is VALT_ PIXEL0D),

and always at reset. Until then, the old value is used.

## 32.14.4. Examples of Use

The use of operator IsFirstPixel is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

  Examples - Demonstration of how to use the operator

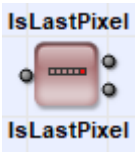
# 32.15. Operator IsLastPixel

**Operator Library: Synchronization**

Operator *IsLastPixel* marks the last pixel of a line (in line mode) / of a frame (in frame mode). The operator can also be used to mark empty lines (in line mode) or empty frames (in frame mode).

The operator has 1 input port and 2 - 3 output ports:

- Input port I,
- Output port O,
- ouput port IsLastO, and
- (optional) output port IsEmptyO.



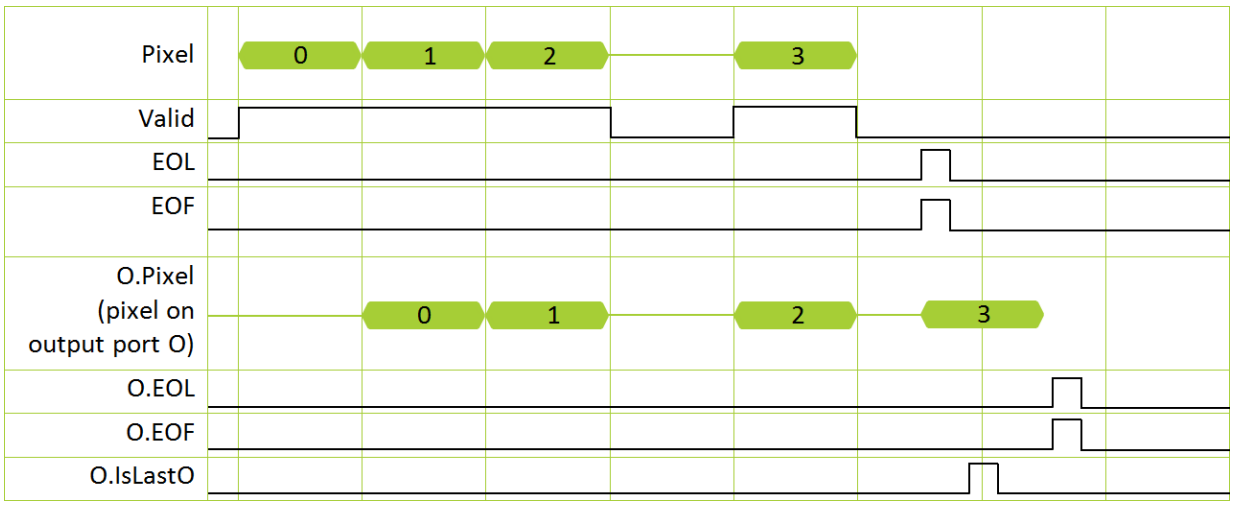

**General Operator Functionality**

Operator *IsLastPixel* always holds the current pixel value. As soon as a new pixel value comes in on input port I, the previous pixel value is put out on output port O. The pixel values received on input port I are not interpreted.

The last pixel of a line (in line mode) or of a frame (in frame mode) is marked with 1 on output port *IsLastO*:

- **Line Mode**: For each EndOfLine (EOL) tag that operator *IsLastPixel* receives, the operator outputs a 1 on its output port *IsLastO*.
- **Frame Mode**: For each EndOfFrame (EOF) tag that operator *IsLastPixel* receives, the operator outputs a 1 on its output port *IsLastO*.

End of Line / End of Frame information is always put out on *IsLastO* with a delay of one or more clock cycles (in accordance to the gap between the last incoming pixel and the incoming end-of-line tag; it's always the number of clock cycles of the gap + 1 additional clock cycle).




**No Empty Lines/Frames allowed**

As long as you do not use IsEmptyO (see below):

- In frame mode, the last line must not be empty.
- In line mode, empty lines are not allowed at all.

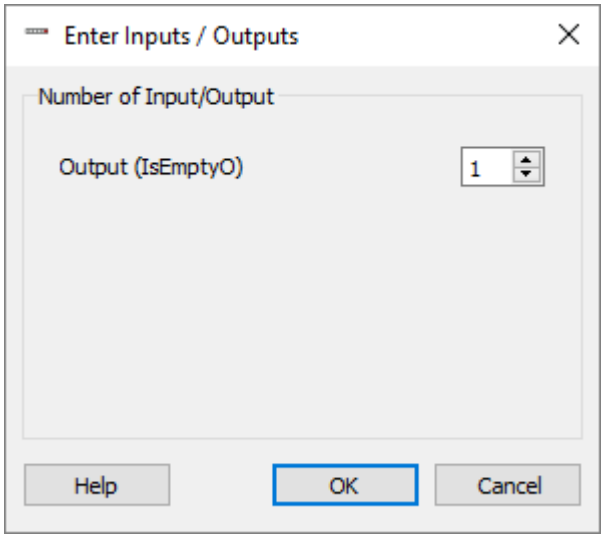
**Marking Empty Lines or Frames (Activating IsEmptyO)**

In addition, you can get a marker for each empty line / each empty frame. The marker will be output on the additional output port *IsEmptyO*.

Output port *IsEmptyO* is de-activated by default.

To activate output port *IsEmptyO*:

1. When instantiating the operator, set the value for Output (IsEmptyO) to 1.




When output port *IsEmptyO* is activated:

- **Line Mode**: For each empty line that operator *IsLastPixel* receives, the operator outputs
  - parallelism x zeros on its output port *O**, and
  - a 1 on on its output port *IsEmptyO*.
- **Frame Mode**: For each empty frame that operator *IsLastPixel* receives, the operator outputs
  - parallelism x zeros on its output port *O*, and
  - a 1 on on its output port *IsEmptyO*.

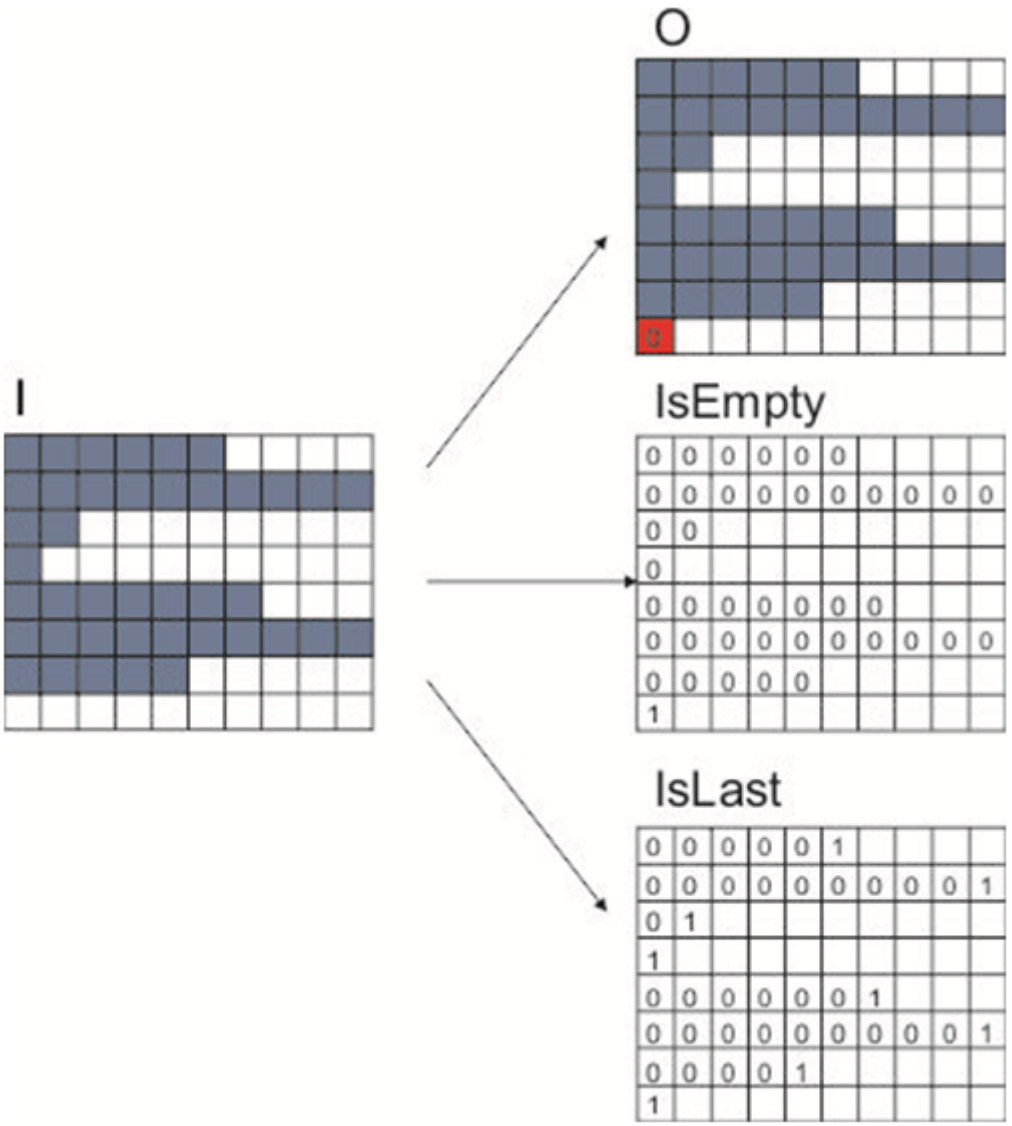
In all other cases, output port *IsEmptyO* is 0.

*Example: If you work in line mode with a parallelism of 6, on output port *O* six pixel are inserted for each empty line.

**Pixel Stops only at Start of New Pixel or End-of-Line Tag**

Operator *IsLastPixel* outputs a pixel only an the moment the operator receives the next pixel or an end-of-line tag (EOL). This is of special importance in cases where the end-of-line tag is not received directly after the last pixel.

Example (parameter *Mode* = Line) with activated output port *IsEmptyO*:




The red pixel on output port O is the dummy pixel with the value 0 that has been inserted for the empty line.

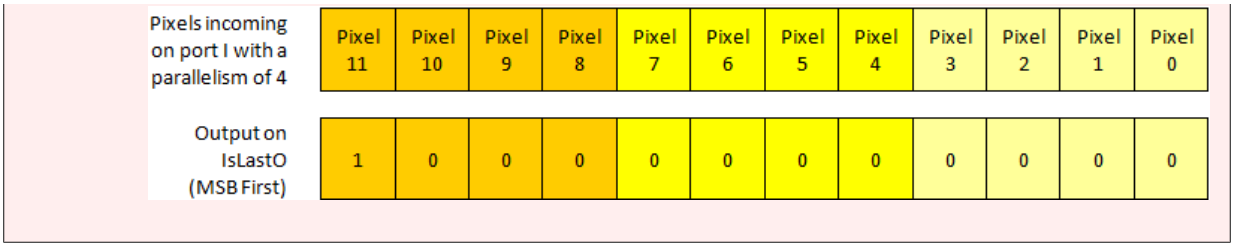
On output port *IsLastO*, the empty line gets a 1, that is, the empty line looks the same as the fourth line of the example which contains one pixel already when incoming on input port I. Thus, when output port *IsEmptyO* is activated, the output on port *IsLastO* can only be interpreted in conjunction with the output on port *IsEmptyO*.

In the example above, port *IsLastO* outputs a 1 for line 4 and for line 8. The 0 for line 4 in *IsEmptyO* tells us that the incoming line was NOT empty and therefore contained 1 pixel. The 1 for line 8 in *IsEmptyO* tells us that the incoming line WAS empty, i.e., contained 0 pixel.

**Parallelism > 1**

If parallelism > 1, only the last pixel within the parallelism (highest bit) is marked with a 1 on output port *IsLastO*. All other pixels of the parallelism are 0.

Example: When Parallelism=4, at the last parallel word the output port *IsLastO* is 0x8, but in all other cases 0:




### 32.15.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, image data input |
| **Output Links** | O, image data output<br>IsLastO, output of 1 for last pixel in line/frame and 0 for all other pixels<br>IsEmptyO (optional), output of 1 for each empty line/frame and 0 for lines/frames containing image data |

### 32.15.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1:63] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_LINE1D, VALT_IMAGE2D} | as I |
| **Color Format** | {VAF_COLOR, VAF_GRAY} | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any❶ | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link IsLastO | Output Link IsEmptyO (optional) |
|---|---|---|
| **Bit Width** | 1 | 1 |
| **Arithmetic** | unsigned | unsigned |
| **Parallelism** | as I | as I |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | as I | as I |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | as I | as I |
| **Max. Img Height** | as I | as I |

❶ The maximum image width must be divisible by the parallelism.

### 32.15.3. Parameters

| Mode | |
|---|---|
| **Type** | static or dynamic write parameter |
| **Default** | Frame |
| **Range** | {Line, Frame} |

If set to *Line*, the operator marks the last pixel in a line. The operator outputs a 1 on its output port *IsLastO* for each last pixel of a line.

If set to *Frame*, the operator marks the last pixel in a frame. The operator outputs a 1 on its output port *IsLastO* for each last pixel of a frame.

You can set this parameter to *static* or *dynamic*. When you use the parameter as a dynamic parameter: A shadow register is implemented, and the change is taken over:

- between two images (when I is VALT_IMAGE2D), or
- between two lines (when I is VALT_ LINE1D), or
- directly (when I is VALT_ PIXEL0D),

and always at reset. Until then, the old value is used.

### 32.15.4. Examples of Use

The use of operator IsLastPixel is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

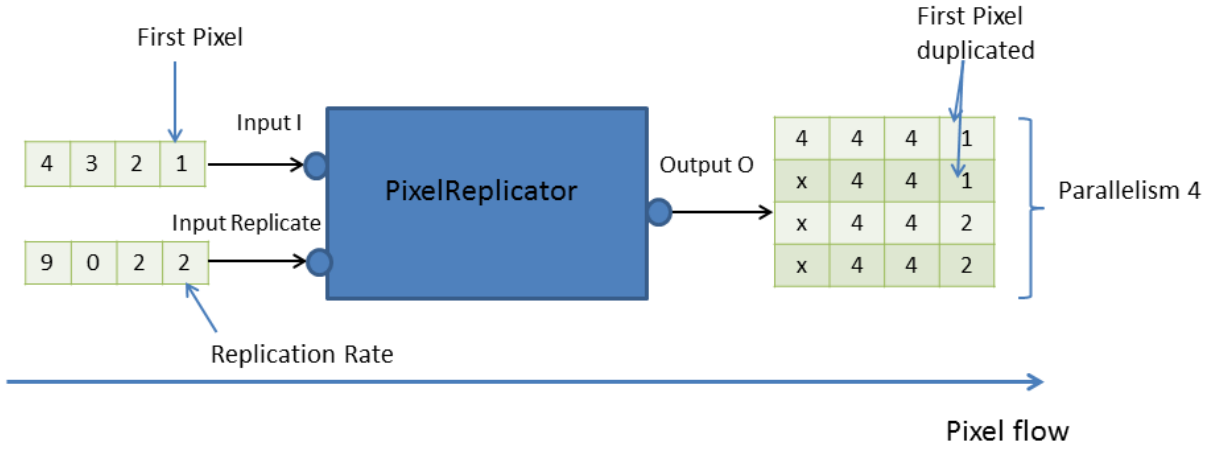
  Examples - Demonstration of how to use the operator

# 32.16. Operator PixelReplicator

**Operator Library: Synchronization**

The M-type operator PixelReplicator replicates the pixels at input link I. The pixel value at input Replicate sets the number of replications for each pixel at input link I. A value of "0" at input link Replicate means that the pixel at input link I will be deleted. The number of pixels at both input links has to be the same. Variable line lengths, empty images and asynchronous inputs are allowed. The parallelism at the input is one. The parallelism at the output can be any.

The following example will explain the functionality of this operator.




Here the parallelism at the output is four, at the input (as always) one. As shown in the figure above pixel "1" at input link I is duplicated as the input at Replicate is "2". The same happens to pixel "2" at input link I. Due to a parallelism of four the duplicated pixels "1" and "2" are forwarded at once to the output link O. Pixel "3" at input link is deleted due to "0" at input Replicate. Pixel"4" shall be replicated 9-times. Since parallelism is smaller than the replication rate the input pipeline is hold until all values are replicated. Non used output pixels are not defined at the output link O (marked with "x" in the figure above).

**Warning**

The image dimensions at input links I and Replicate have to be the same. Otherwise the simulation gives an error. If the operator is used at this specification violation during runtime of the applet in hardware, the operator behavior is undefined. Ensure that both inputs transport exaclty the same number of pixel. If this is not possible use a *SYNC* operator before.

## 32.16.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, data which has to be replicated<br>Replicate, sets the replication rate |
| **Output Link** | O, data output |

## 32.16.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Replicate | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64] | [1, 31]❶ | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | 1 | 1 | any |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | 2D, 1D, 0D | as I | as I |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any❷ | as I | I*((2^W❸)-1) ❹ |
| **Max. Img Height** | any | as I | as I |

❶❷ with Replicate.BitWidth + log2(I.MaxImageWidth) <= 31.
❸ Bit width at input Replicate
❹ Rounded up to the next multiple of the parallelism.

**Synchronous and Asynchronous Inputs**

- Both inputs can be asynchronous to each other.

## 32.16.3. Parameters

None

## 32.16.4. Examples of Use

The use of operator PixelReplicator is shown in the following examples:

- Section 12.9.3.2.3, 'Geometric Transformation using **PixelReplicator**'

  Examples- Geometric Transformation using **PixelReplicator**

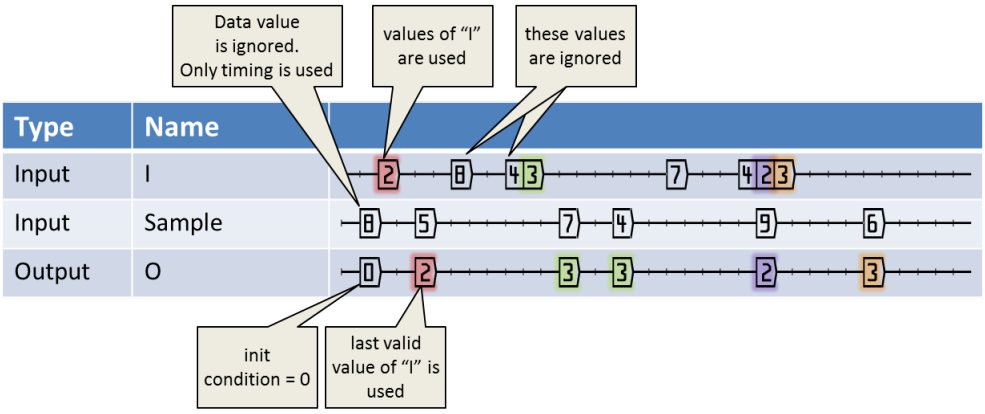
# 32.17. Operator PixelToImage

**Operator Library: Synchronization**

The operator PixelToImage allows to synchronize the 0D pixel stream arriving at input link I with an 0D, 1D or 2D image at input link Sample. You can imagine the output images at link O as a combination of the input I and Sample. While O follows Sample in timing and frame rate, the data content is copied from I. The pixel values at the Sample input are not used in the operator only the timing is used. Input I is fully asynchronous from input Sample. Flow control does not influence I. The input I can never be blocked.

This operator is useful for adding time stamps and other information into images.

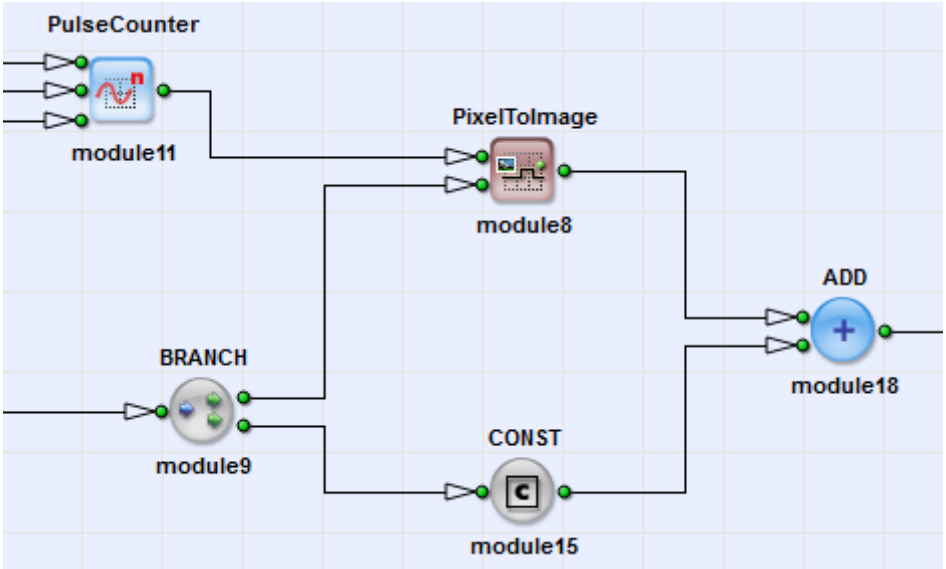
Since input link I is asynchronous, the output link will always forward the last valid data from input I. When a sample input arrives before a valid input I is present, then the output is set to zero. Check the following waveform for clearification.




**The operator can be used like an O operator:**

The operator PixelToImage is a P type operator. According to the rules of links, when several links are merged after a P type operator, a SYNC operator has to be used.

However, operator PixelToImage is different: Between input "Sample" and output "O" (path sample -> O), the operator is handled like an O type operator by VisualApplets. This means in effect a pipeline equalization is carried out. For a merge, no synchronization is required.



**Behavior During Design Simulation**

During design simulation, the operator *PixelToImage* discards all data on input "I" since the exact timing cannot be simulated. Instead, the pixel values of the generated output data are defined via simulation parameter *SimulationPixelValue*, while the dimension of the output data is set according to the data on port "Sample".

### 32.17.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I, data which has to be sampled<br>Sample, provides the timing |
| **Output Link** | O, data output |

### 32.17.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Sample | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | [1, 64]❷ | as I |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} | as I |
| **Parallelism** | 1 | 1 | as Sample |
| **Kernel Columns** | 1 | 1 | as Sample |
| **Kernel Rows** | 1 | 1 | as Sample |
| **Img Protocol** | VALT_PIXEL0D | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | as Sample |
| **Color Format** | any | any | as I |
| **Color Flavor** | any | any | as I |
| **Max. Img Width** | any | any | as Sample |
| **Max. Img Height** | any | any | as Sample |

❶❷ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

**Synchronous and Asynchronous Inputs**

- All inputs are asynchronous to each other.

### 32.17.3. Parameters

None

### 32.17.4. Examples of Use

The use of operator PixelToImage is shown in the following examples:

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 13.3, 'Functional Example for Specific Operators of Library Memory and Library Signal'

  Examples - Demonstration of how to use the operator

# 32.18. Operator RemoveImage

**Operator Library: Synchronization**

The operator RemoveImage is used to completely remove images. The control whether an image is removed is made using the binary input link *Rem*. If the very first pixel of *Rem* is value 1, the image is removed. If the pixel is 0, the image is forwarded to the output.

Both inputs have to be synchronous, i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

Moreover, empty images or images with an empty first line will always be removed!

## 32.18.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I, data input<br>Rem, control input |
| **Output Link** | O, data output |

## 32.18.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Rem | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D | as I |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | any | as I |
| **Max. Img Height** | any | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

**Synchronous and Asynchronous Inputs**

- All inputs are synchronous to each other i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

## 32.18.3. Parameters

None

## 32.18.4. Examples of Use

The use of operator RemoveImage is shown in the following examples:

- Section 3.6.4, 'M-type Operators with Multiple Inputs'

  Synchronization Rules - Using an M-type Operator with Synchronous Inputs

- Section 9.2, ' Multiple DMA Channel Designs '

  Remove 9 out of 10 images.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

# 32.19. Operator RemoveLine

**Operator Library: Synchronization**

The operator RemoveLine is used to completely remove image lines. The control whether a line is removed is made using the binary input link *Rem*. If the very first pixel of each line at input link *Rem* is value 1, the line will be removed. If the pixel value is 0, the line is forwarded to the output.

Both inputs have to be synchronous i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

Empty lines cannot transport the information whether they have to be removed or not as they have no data pixel. In this case, the lines will always be removed. If all lines of an image are removed, the operator will output an empty image at its output.

## 32.19.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I, data input<br>Rem, control input |
| **Output Link** | O, data output |

## 32.19.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Rem | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_IMAGE1D} | as I | as I |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | any | as I |
| **Max. Img Height** | any | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

**Synchronous and Asynchronous Inputs**

- All inputs are synchronous to each other i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

## 32.19.3. Parameters

None

## 32.19.4. Examples of Use

The use of operator RemoveLine is shown in the following examples:

- Section 3.6.7, 'Bandwidth Bottlenecks'

  Bandwidth Bottlenecks - Reducing the parallelism after the removal of pixels.

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.9.2, 'Downsampling 3x3'

  Examples - Downsampling by factor 3x3 without the use of operator *SampleDn*.

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

# 32.20. Operator RemovePixel

**Operator Library: Synchronization**

The operator *RemovePixel* removes pixels between the input link I and output link O in correspondence to the control input link *Rem*. The remove input link *Rem* is a binary input link which controls the removal of pixels. If a pixel at *Rem* is set to one, the pixel at the same position at input link I is removed. The following pixels are shifted left. Thus, the output lines are reduced in their width by the number of removed pixels.

Due to the removing of pixels it is possible to construct lines with a line width which is not divisible by the parallelism of the link.

To generate lines lengths at output link *O* that are divisible by the parallelism of the link, the operator inserts dummy pixels. The content of these dummy pixels is undefined (VA-simulation sets these value to 0).

You have two options for creating lines with a length divisible by the link parallelism:

You can make the operator either

- fill the lines in question with dummy pixels until the line length is divisible by the link parallelism (see example 1), or
- shift the pixels that don't make up a full block of parallel pixels to the next line (see example 2). In this case, the operator inserts dummy pixels only in the last line of the frame if necessary.

You control this behaviour of the operator by parameter *FlushCondition*: If the parameter is set to **EoL** (End of Line), the operator will add the dummy pixels to the end of the lines which are not divisible by the parallelism. The line structure of the image will be preserved, even if all pixels are removed from the image (the image than consists of only empty lines). If all pixels are removed in the frame, the frame will consist of empty lines only. The number of lines is not changed.

If parameter *FlushCondition* is set to **EoF** (End of Frame): If the last pixels of a line do not complete a block of parallel pixels, these last pixels are moved to the next output line. Thus, the lines end at parallelism boundaries and the exceeding pixels are moved to the following line. The operator will therefore only create dummy pixels at the end of the frame. The line structure may be changed in this mode and the last line may be longer (by 1 parallelism). Therefore, the MaxWidth is 1 times parallelism wider than the input link. The EoF mode is often used for image compression applications.



**Resource Consumption**

The operator's FGPA resource consumption strongly increases with the parallelism. Basler recommends to use low parallelism at this operator.

The parallelism can be reduced by shifting the parallel pixel into a kernel (i.e. by using *CastKernel*) before using the *RemovePixel* operator.

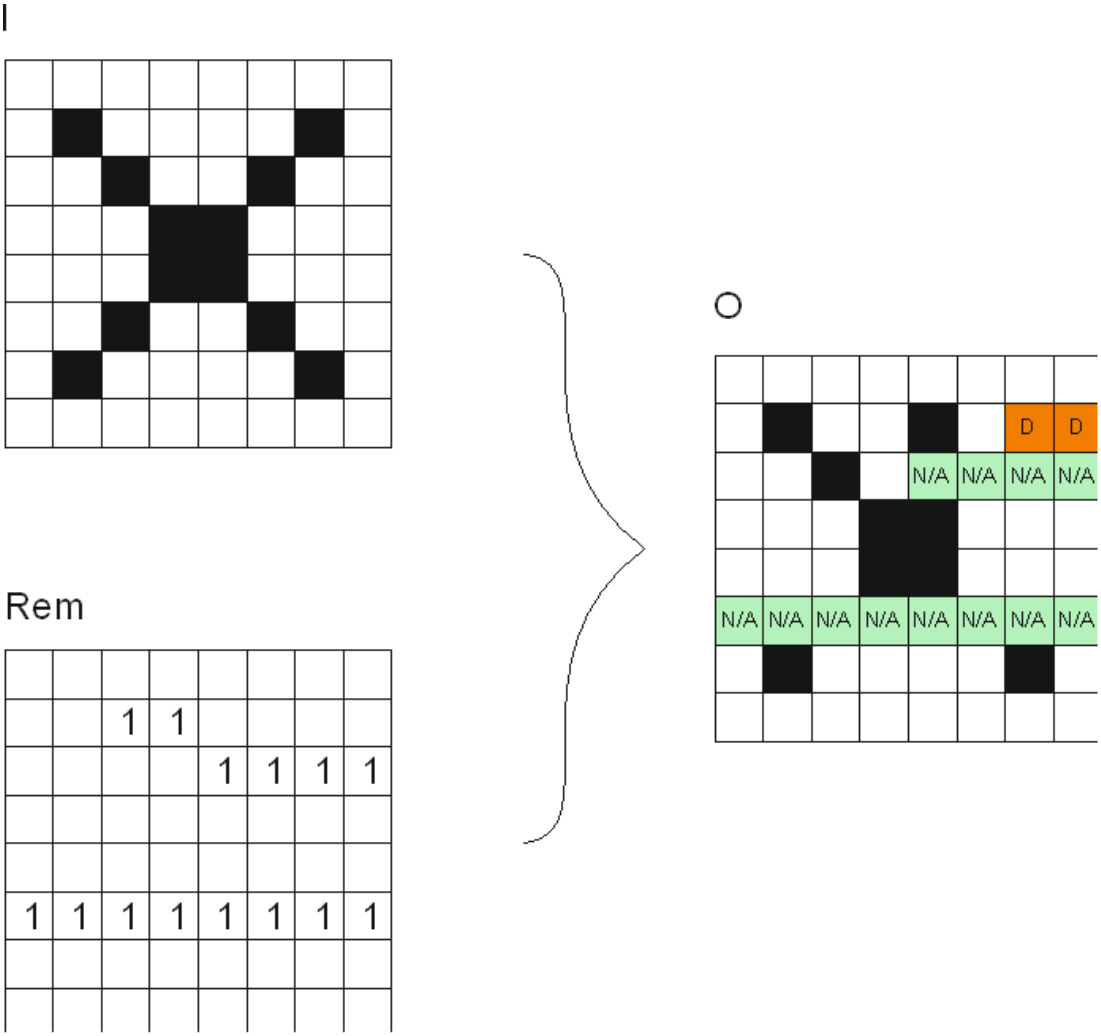
If you want to remove only the last pixels of a line, you can use the operator *CutLine* in the mode *truncate* instead. This saves resources.

---

**Example 1** (parameter *FlushCondition* is set to **EoL**):

In this example, *FlushCondition* is set to **EoL** (default). The operator is used with a parallelism of 4 and gets the images as shown in the picture.

In the second line, two pixels are removed. This will cause the following pixels to be shifted. Since 6 (the new line length) is not dividable by 4 (parallelism), 2 dummy pixels are inserted. In the third line, 4 pixel are removed. Since 4 (the new line length) is dividable by 4, the line simply gets shorter and no dummy pixels are inserted. The 6th line is removed completely. For the following simulation it is treated as an empty line.

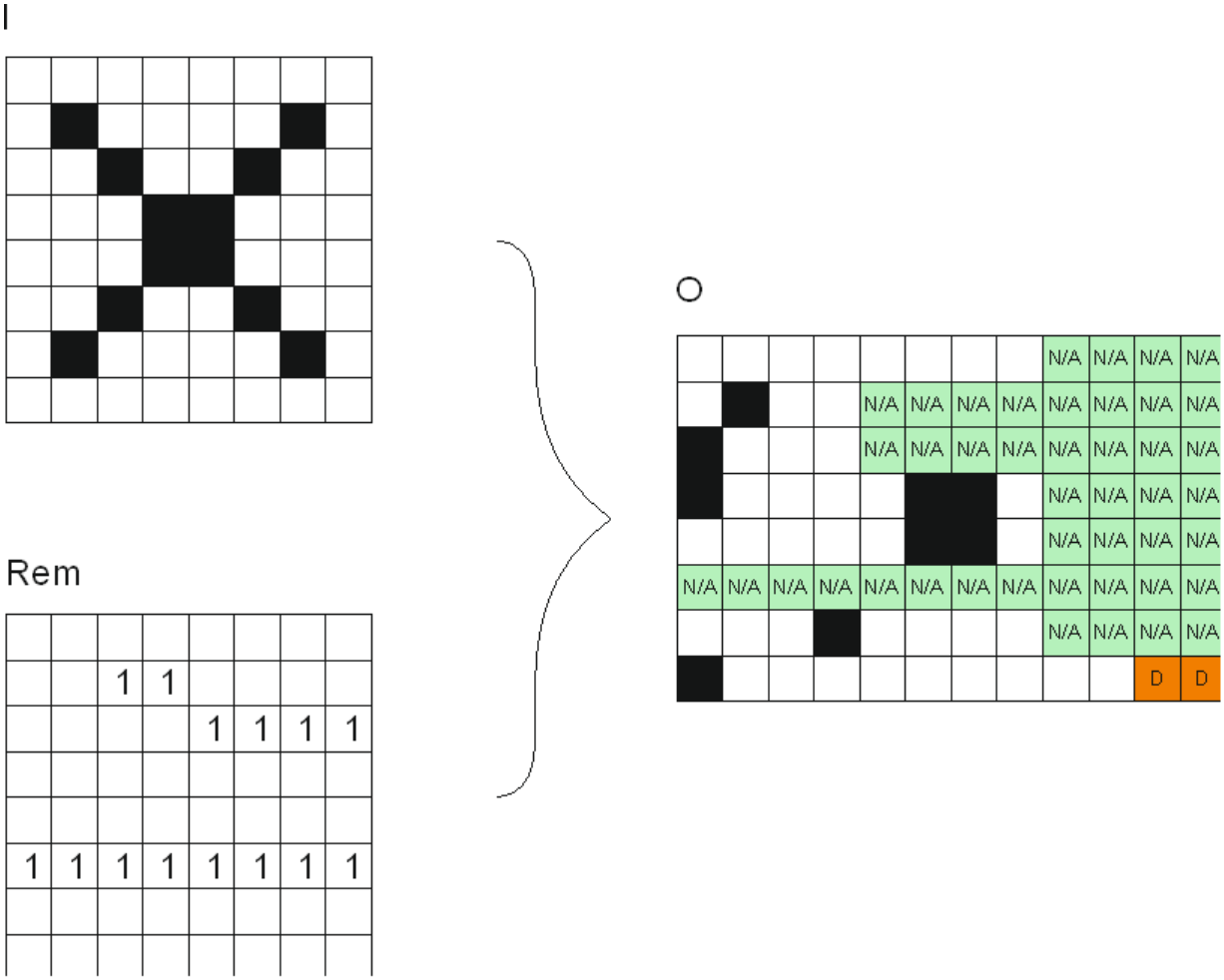



Example 1: Behavior if FlushCondition is set to EoL and parallelism is set to 4 (Rem is always 0, except for the pixels set to 1)

---

**Example 2** (parameter *FlushCondition* is set to **EoF**):

In this example, *FlushCondition* is set to **EoF**. The operator is used with a parallelism of 4 and gets the images as shown in the picture.




Example 2: Behavior if FlushCondition is set to EoF and parallelism is set to 4 (Rem is always 0, except for the pixels set to 1)

**Changed Maximum Image Width in EoF Mode - Be attentive with Older Designs!**

The maximum image width has been corrected for EoF mode to be 1 parallelism wider (for Parallelism > 1). Please look at the example to understand the issue. This might cause DRC errors in older designs.

If you know for sure that this extra width is not needed, you may use *SetDimension* to set the dimension back to the maximum input width.

## 32.20.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Links** | I, data input<br>Rem, control input |
| **Output Link** | O, data output |

## 32.20.2. Supported Link Format

| Link Parameter | Input Link I | Input Link Rem | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | 1 | as I |
| **Arithmetic** | {unsigned, signed} | unsigned | as I |
| **Parallelism** | any | as I | as I |
| **Kernel Columns** | any | 1 | as I |
| **Kernel Rows** | any | 1 | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | as I | as I |
| **Color Format** | any | VAF_GRAY | as I |
| **Color Flavor** | any | FL_NONE | as I |
| **Max. Img Width** | any | as I | as I (EoL mode) / as I + parallelism (EoF mode) |
| **Max. Img Height** | any | as I | as I |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

**Synchronous and Asynchronous Inputs**

- All inputs are synchronous to each other i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

## 32.20.3. Parameters

| FlushCondition | |
|---|---|
| **Type** | static parameter |
| **Default** | EoL |
| **Range** | {EoL, EoF} |

This parameter controls the insertion of dummy pixels. In EoL mode, dummy pixels will be inserted at the end of a line if the line length is not dividable by parallelism.

In EoF mode, if the last pixels of a line do not complete a block of parallel pixels, these pixels are moved to the next output line. Thus, the lines end at parallelism boundaries and the exceeding pixels are moved to the following line. The operator will therefore only create dummy pixels at the end of the frame if the last line of the frame is not dividable by the parallelism. The MaxWidth of the frame at the output link O is the frame width at the Input-Link + 1 parallelism.

The parameter is deactivated if image protocol VALT_PIXEL0D is used. For the image protocol VALT_LINE1D, only EoL can be used.

## 32.20.4. Examples of Use

The use of operator RemovePixel is shown in the following examples:

- Section 11.2, 'Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Penta Platform'

  Examples - Demonstration of how to use the operator

- Section 11.3, 'Functional Example for the *FrameBufferMultRoi* User Library Element on the imaFlex CXP-12 Quad Platform'

Examples - Demonstration of how to use the operator

- Section 12.1.8, 'Run Length Encoder'

  Examples - A run length encoding example of defined format.

# 32.21. Operator ReSyncToLine

**Operator Library: Synchronization**

The operator ReSyncToLine is used to re-synchronize calculation results with the content of the current line. Thus, calculation results which are available at the end of a line can be applied to the same line. This allows the implementation of multiple-pass algorithms at the line level.

ReSyncToLine fetches the last pixel of each line on all k input links PI[k]. Also the current line at input link I is stored. After the last line pixel of the inputs is processed in the operator, the content of the line is output at the link O. The other output links PO0..PO63 output the fetched values of PI0..PI63. The PO values are held constant. All PO and O output ports are synchronous.

All input links (I and PI0..PI63) have to be fully synchronous, i.e. they have to be sourced by the same M-type operator through an arbitrary network of O type operators.

The operator delays the output by one line.

## 32.21.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I, image input<br>PI[k], data input |
| **Output Links** | O, image output<br>PO[k], data output |

## 32.21.2. Supported Link Format

| Link Parameter | Input Link I | Input Link PI[k] |
|---|---|---|
| **Bit Width** | [1, 64]❶ | [1, 64]❷ |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | any |
| **Kernel Rows** | any | any |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D} | as I |
| **Color Format** | any | any |
| **Color Flavor** | any | any |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

| Link Parameter | Output Link O | Output Link PO[k] |
|---|---|---|
| **Bit Width** | as I | as PI[k] |
| **Arithmetic** | as I | as PI[k] |
| **Parallelism** | as I | as I |
| **Kernel Columns** | as I | as PI[k] |
| **Kernel Rows** | as I | as PI[k] |
| **Img Protocol** | as I | as I |
| **Color Format** | as I | as PI[k] |
| **Color Flavor** | as I | as PI[k] |
| **Max. Img Width** | as I | as I |

| Link Parameter | Output Link O | Output Link PO[k] |
|---|---|---|
| **Max. Img Height** | as I | as I |

⓬ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

**Synchronous and Asynchronous Inputs**

- All inputs are synchronous to each other i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

## 32.21.3. Parameters

None

## 32.21.4. Examples of Use

The use of operator ReSyncToLine is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

  Examples - Demonstration of how to use the operator

# 32.22. Operator RxImageLink

**Operator Library: Synchronization**

The *RxImageLink* operator is used to receive images from a *TxImageLink* operator in the same design. Both operators establish a connection without a link. The image format remains the same, i.e., the format at the ouput of RxImageLink is the same as the format *TxImageLink* receives at its input.

With the image transfer between the *TxImageLink* and *RxImageLink* operators, it is possible to implement loops in a design.

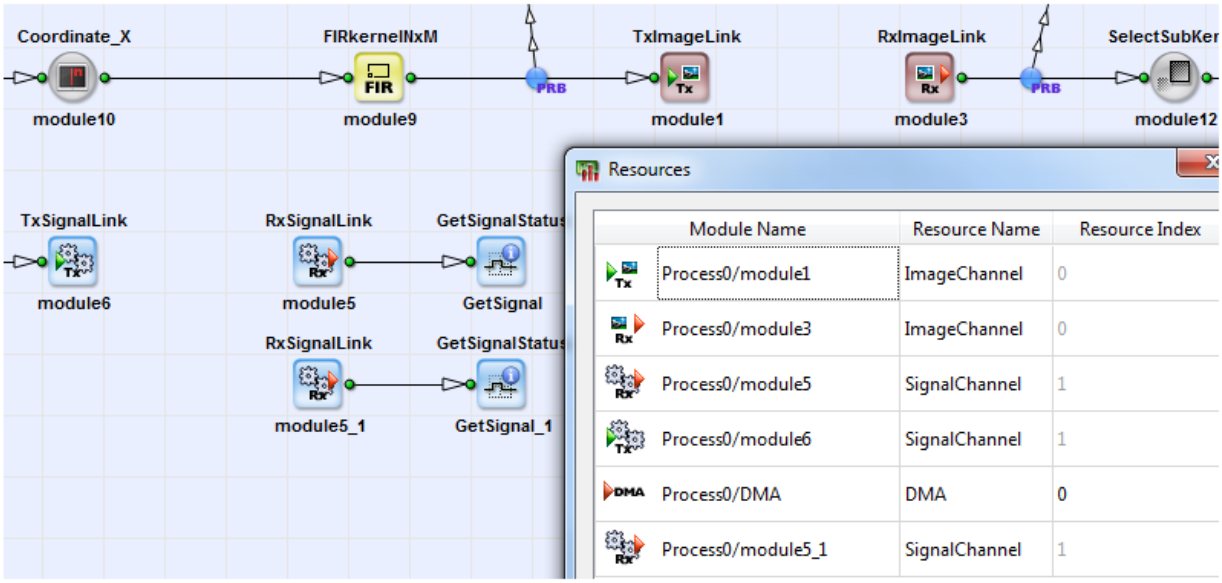
**Loops require data buffering strategy**

Operators *TxImageLink* and *RxImageLink* do not buffer data. Therefore, when implementing loops, you need to take special care with regard to data buffering to avoid deadlocks.

The parameter *Channel_ID* defines a channel ID to address the sending *TxImageLink* operator. The parameter value has to be unique and must not be used by any other RxImageLink operator in the design.

The parameter value has to match with the *Channel_ID* of one of the *TxImageLink* operators in the same design.

Each *TxImageLink* operator in a design is connected to exactly one *RxImageLink* operator via one channel ID. In the *Resource Dialog* of VisualApplets, you can see that one *ImageChannel* resource is used for each *TxImageLink-RxImageLink* connection. Resource *ImageChannel* allows to control the assignment of individual *TxImageLink* operators to individual *RxImageLink* operators. For the number of available *ImageChannel* resources (which also defines the maximum number of allowed *TxImageLink* and *RxImageLink* operators in a design), see Appendix A, '*Device Resources*'.

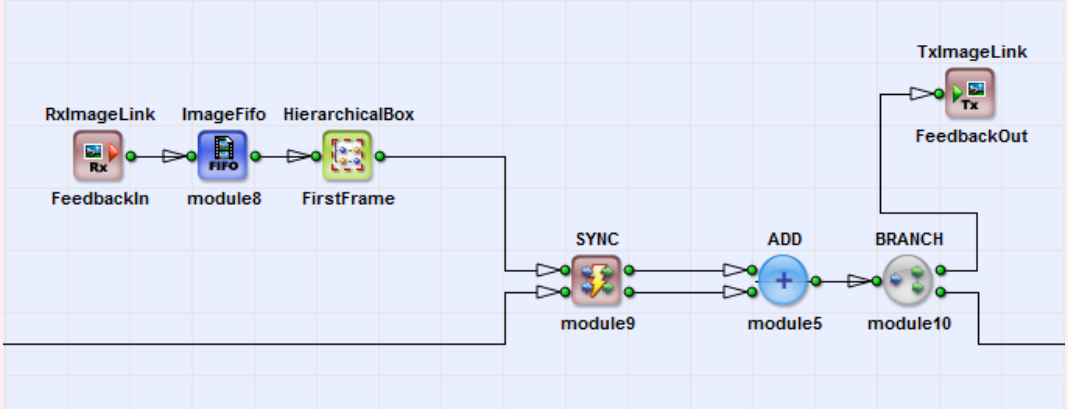



**Parametrization of Link Format in Loops**

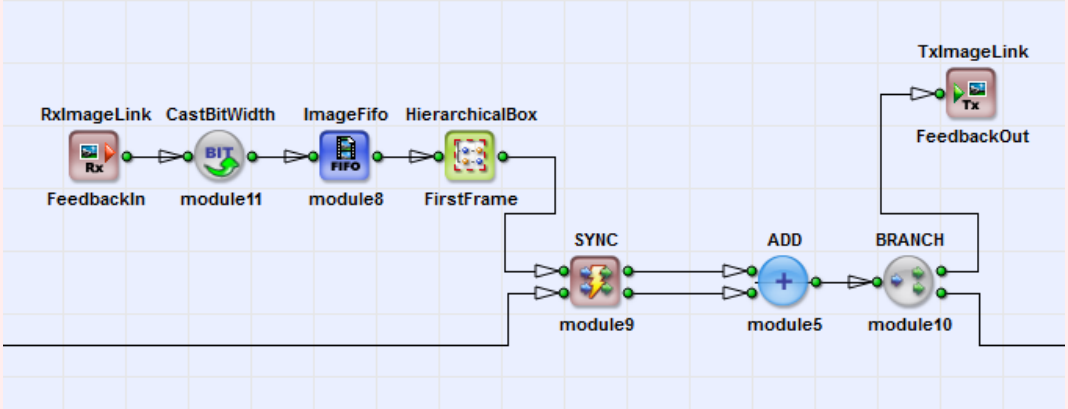
As soon as the link properties dialog is opened, the "automatic update" feature will adapt the link properties (such as bit width, or image dimensions) according to the operator chain's configuration. The input format of *TxImageLink* always defines the output format of *RxImageLink* (100% automatic consistency). Therefore, when you use operators *TxImageLink* and *RxImageLink* to implement loops, you need to take special care regarding the parametrization of the link formats.

**Example**

Wrong Implementation, value of link parameter *Bit Width* gets higher with every iteration through the loop:




Right Implementation, Operator CastBitWidth changes the value of link parameter *Bit Width* back to the original value with every iteration through the loop:




**Synchronizing Channels between different hierarchical design levels**

If you use a TxImageLink/RxImageLink pair to set up a data transfer channel between different hierarchical design levels, this connection is treated as an M operator. Therefore, you may need to implement additional synchronization elements (that are not required with a direct connection.)

### 32.22.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 32.22.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

### 32.22.3. Parameters

| Channel_ID | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0,1023] |
| The channel ID of the image link. See descriptions above. | |

### 32.22.4. Examples of Use

The use of operator RxImageLink is shown in the following examples:

- Section 11.4, 'Functional Example for Loading Test Images Using *ImageInjector* '

  Examples - Demonstration of how to use the operator

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

- Section 12.12.2, 'Depth From Focus Using Loops'

  Examples - Depth From Focus using Loops

# 32.23. Operator SourceSelector

**Operator Library: Synchronization**

The operator works as a switch between up to 64 M-Type sources. It can be used to switch between alternative implementations. You can use it, for example, to switch between different camera ports, or between different DMA inputs.

The selection is controlled by parameter *SelectSource*. The switching between two input ports keeps the granularity of the currently transmitted frame and line, i.e., the current frame being transmitted will be completely transmitted before switching to a new input port. In other words, the operator will not switch to another input while a frame is processed.

The operator acts like a Trash operator for all links which are currently not selected.

Note that all input links must be in the same format except the maximal image width and height.

The source type is specified by the parameter *InfiniteSource*. Setting it to ENABLED allows the operator to be connected to camera sources. Setting it to DISABLED allows the operator to be used between normal VA operators subsequent to a buffer.

We recommend not to use the *SourceSelector* operator for switching between sources which are sourced by the same M-type source. In such cases, use the *IF* or the *CASE* operator instead since these operators use much less resources than the *SourceSelector* operator.

**Operator Restrictions**

- Empty frames are not supported.
- Empty lines are not supported.

## 32.23.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I[0], data input<br>I[n-1], data input |
| **Output Link** | O, data output |

## 32.23.2. Supported Link Format

| Link Parameter | Input Link I[0] | Input Link I[n-1] | Output Link O |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | as I[0] | as I[0] |
| **Arithmetic** | {unsigned, signed} | as I[0] | as I[0] |
| **Parallelism** | any | as I[0] | as I[0] |
| **Kernel Columns** | any | as I[0] | as I[0] |
| **Kernel Rows** | any | as I[0] | as I[0] |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | as I[0] | as I[0] |
| **Color Format** | any | as I[0] | as I[0] |
| **Color Flavor** | any | as I[0] | as I[0] |
| **Max. Img Width** | any | any | auto❷ |
| **Max. Img Height** | any | any | auto❸ |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❷ The maximum output image width is the maximum of all input widths.
❸ The maximum output image height is the maximum of all input heights.

### 32.23.3. Parameters

| SelectSource | |
|---|---|
| **Type** | static/dynamic read/write parameter |
| **Default** | 0 |
| **Range** | [0, n-1] |
| This parameter selects the port and forwards its image data to the output. The data at all other ports is discarded. | |

| InfiniteSource | |
|---|---|
| **Type** | static parameter |
| **Default** | DISABLED |
| **Range** | {ENABLED, DISABLED} |
| When set to ENABLE, the operator allows its direct connection to infinite sources like Camera operators without the need of a buffer. | |

### 32.23.4. Examples of Use

The use of operator SourceSelector is shown in the following examples:

- Section 9.3.1, ' Synchronizing Cameras '

  Operator used to switch between two cameras.

- Section 12.6.1, 'Hardware Test'

  An example for hardware self test of DMA, RAM, GPIOs, Trigger and LEDs.

- Section 12.6.2, 'Image Dimension Test'

  Example - The image dimension is measured and can be used to analyze the design flow.

- Section 12.6.4, 'Manual Image Injection'

  Example - For debugging purposes images can be inserted manually.

- Section 12.6.5, 'Image Monitoring'

  Example - For debugging purposes image transfer states on links can be investigated.

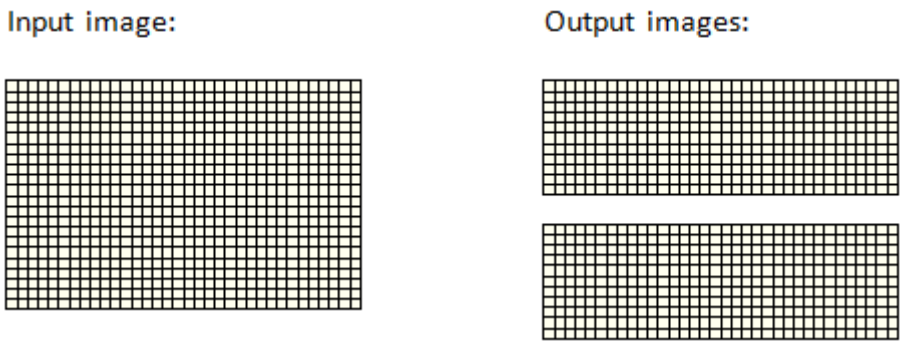
- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

# 32.24. Operator SplitImage

**Operator Library: Synchronization**

This operator splits the input image into a number of smaller images. The image height of the smaller images is specified by parameter *ImageHeight*. For example, setting *ImageHeight* to 512 and using input images of 1024 lines, the operator will split the input image into two new smaller images with the image height of 512.

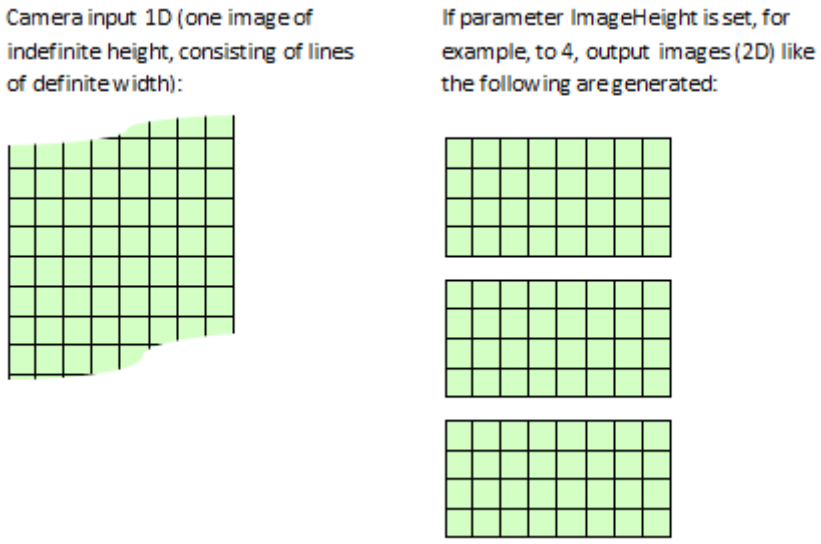



If the height of an input image is not divisible by the *ImageHeight* parameter value, the input image will be split into images of the specified height as far as possible. The remaining lines will generate an output image of smaller height. For example, if an input image of height 1024 is split into chunks of height 400, the operator will output two images of height 400 and and one image of height 224. If the input image height is less than parameter *ImageHeight*, the operator will not split the input image.

**Increased Frame Rate**

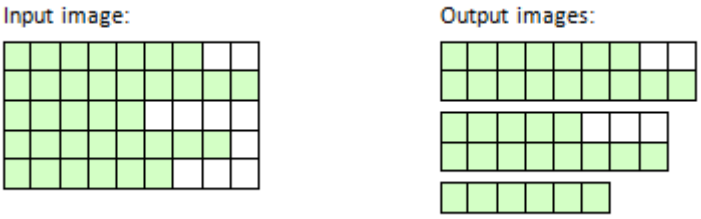
Note that the operator increases the frame rate by keeping the original bandwidth.

SplitImage can be used to perform a 1D to 2D conversion. The conversion is automatically performed if a link using the image protocol VALT_LINE1D is connected to the operator's input. For example, the 1D input of a line scan camera can be split into images of a specified height, effectively creating a 2D output stream. Example:




When changing the ImageHeight value dynamically while acquisition is running, the operator guarantees image integrity. The operator keeps the old ImageHeight value until the output frame is finished. After the completion of the current output frame the operator will start using the new ImageHeight value for further splitting.

The operator supports variable line length for all ImageHeight values, for example ImageHeight = 2:




The end of each line is marked by a Eol (End of Line) flag, so that there are no undefined data in the output images.

## 32.24.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | P |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 32.24.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | VALT_IMAGE2D |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | any |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

## 32.24.3. Parameters

| ImageHeight | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 1 |
| **Range** | [1, Maximum Output Image Height] |
| Maximum number of lines in the output image. | |

## 32.24.4. Examples of Use

The use of operator SplitImage is shown in the following examples:

- Section 10.2.2, 'GigE Vision Line Scan Cameras'

  Tutorial - Basic Acquisition

- Section 12.3.1, 'Blob 1D'

  Examples - Shows the usage of operator *Blob_Analysis_1D* in line scan applications.

- Section 12.4.2.5, 'Color Plane Separation Option 5 - Sequential Output with Advances Processing'

  Example on separation of color planes. The RGB input is split into its component and sequentially output via one DMA channel. The splitting if performed by collecting same components in parallel words and reading with FrameBufferRandomRead.

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

- Section 12.9.4, 'ImageSplitAndMerge'

  Examples - Shows how to split an merge image streams. Appends a trailer to the image.

- Section 12.15.8.1, 'Line Scan Trigger for microEnable IV VD4-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.9.1, 'Line Scan Trigger for microEnable IV VQ4-GE/-GPoE Using Signal Operators and Operator **ActionCommand**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation. The camera can be triggered via cable connection to the trigger ports or directly with ActionCommands. See here especially the documentation Section 29.3, 'ActionCommand'. See also the Action Command SDK example under SDK_Examples\gbe \ActionCommands in your Framegrabber SDK installation folder.

- Section 12.15.10.1, 'Line Scan Trigger for microEnable 5 marathon/LightBridge VCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.11.1, 'Line Scan Trigger for microEnable 5 VD8-CL/-PoCL Using Signal Operators and Operator **CameraControl**'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.12.1, 'Line Scan Trigger for microEnable 5 marathon VCX QP Using Signal Operators'

  A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.13.1, 'Line Scan Trigger for imaFlex CXP-12 Quad Using Signal Operators'

  A line scan trigger for CoaXPress12 is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

- Section 12.15.14.1, 'Line Scan Trigger for microEnable 5 VQ8-CXP6 Using Signal Operators'

A line scan trigger is presented. The trigger includes an image trigger using a capture gate as well as a multi functional line trigger. External sources, an internal frequency generator or software trigger pulses can be used for trigger generation.

# 32.25. Operator SplitLine

**Operator Library: Synchronization**

This operator splits the input lines into a number of shorter lines. The output line width is specified by parameter *LineLength*. For example, setting *LineLength* to 512 with an input line length of 1024 pixels, will result in two new shorter lines per 1 input line with the length of 512.

If the length of the input line is not divisible by the LineLength parameter value, the input line will be split into lines of the specified length as far as possible. The remaining pixel result in a shorter line width. For example, an input line of 1024 pixels is split into sub lines of length 400, the operator will output 2 lines of the length 400 and the last 3rd line of the length 224.

Note that the operator might generate output images with multiple line lengths. Not all VisualApplets operators can process images using varying line lengths.

The SplitLine operator can also be used to perform a 0D to 1D conversion. Connect a link using the image protocol VALT_PIXEL0D to perform the conversion.

If the value of *LineLength* is changed dynamically during image acquisition, the operator will keep the old LineLength value internally until the currently being processed line is provided at the output link O. After the line is completed, the next split operation will use the new LineLength value.

## 32.25.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

## 32.25.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D} | auto❷ |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | any❸ |
| **Max. Img Height** | any | any❹ |

❶ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].
❷ The output image protocol is the same as the input image protocol if VALT_IMAGE2D or VALT_LINE1D is used. If VALT_PIXEL0D image protocol is used at the input, the operator performs a conversion to the VALT_LINE1D image protocol.
❸ The output maximum image width has to be less or equal than the input maximum image width and greater or equal than parameter *LineLength*.
❹ The output maximum image height has to be grater or equal than the input image height.

### 32.25.3. Parameters

| LineLength | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | 1 |
| **Range** | [Input Parallelism, Maximum Output Image Width], step size = Input Parallelism |
| Maximum line length of the split output lines. | |

### 32.25.4. Examples of Use

The use of operator SplitLine is shown in the following examples:

- Section 3.6.9, 'Infinite Sources / Connecting Cameras'

  Infinite Sources - Connecting operators to cameras. (DRC2 Latency Error)

- Section 12.6.6, 'Image Grayscale Scope'

  Example - For debugging purposes the Scope operator provides options for analyzing gray-scale pictures. .

# 32.26. Operator SYNC

**Operator Library: Synchronization**

The operator SYNC performs time and image dimension synchronization of all input links. The number of links which have to be synchronized is specified upon operator instantiation. The output images on all output links are synchronous, i.e., they are output at the same time and they all have the same image dimension. Thus, the operator synchronizes asynchronous links so that they are O-synchronous and can be used in an O-type VisualApplets operator network.

The operator does not support empty lines or empty frames.

In case of mixed domain synchronization like 2D with 1D and 0D streams, the operator converts the output links to the highest input domain. When using SYNC to synchronize a 2D link with a 1D link, both output links will be set to the 2D format. An error will be issued if the 1D link doesn't provide enough lines. When using the operator to synchronize 1D links with 0D links, all output links will be in the 1D format. In this case, an error will be issued if the 0D link doesn't provide enough pixels.

In case of input images with different image dimensions, the operator synchronizes the images so that all output images have the same size.



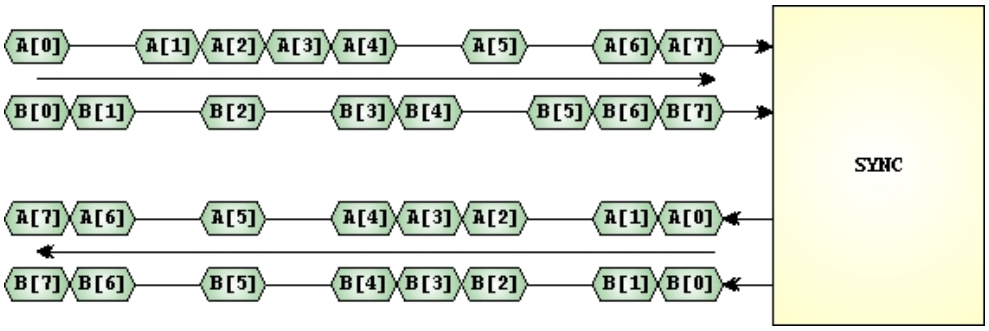
**Synchronization Rules**

Before using the SYNC operator, ensure that you understand all synchronization rules. User manual and Tutorial explain the synchronization rules in detail, see:

- Section 3.6, 'Rules of Links' (User Manual)
- Section 9.3, ' Synchronization of Asynchronous Image Pipelines ' (Tutorial)

Both, the timing synchronization and the image dimension synchronization, are explained in the following.

## 32.26.1. The Timing Synchronization



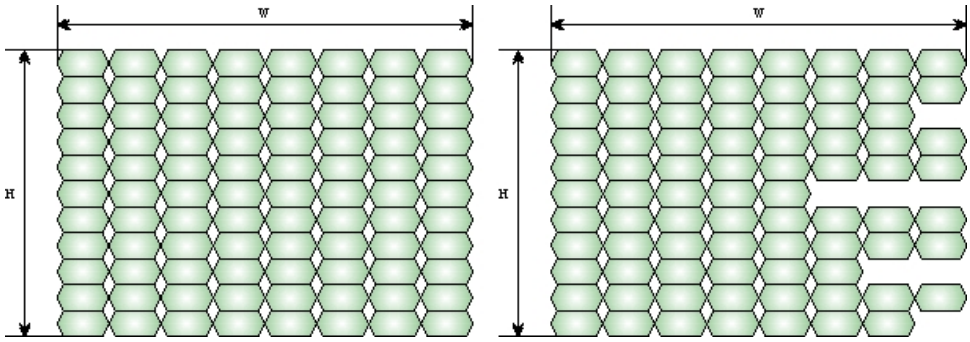

During transmission of the pixels, pauses can occur due to the flow control or the link bandwidth exceeding the bandwidth of the image source. The figure above shows how the time synchronization is performed for 2 links. Both input links have a different timing. However, the output link data is synchronized, i.e., the pixels are output at the same time on all links.

The operator acts like a valve for all input links: The input links are closed until a valid pixel is present on all inputs. The operator will then forward the pixels of all inputs to the output, i.e., open the valve.

## 32.26.2. The Image Dimension Synchronization

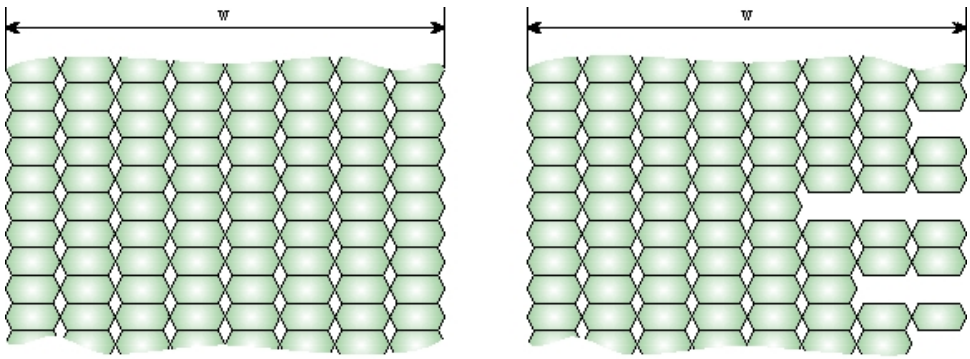
The SYNC operator is capable of synchronizing images using different image protocols VALT_IMAGE2D, VALT_LINE1D or VALT_PIXEL0D. The following three figures illustrate some images of the different image protocols.

- 2D




2D images have a finite height H and a finite width W. The first image shown in the figure is a regular 2D image. The 2nd image is an irregular 2D image having different line lengths.

- 1D




1D lines have a finite width W but an infinite height. Images of these types are usually created in line scan applications. Again, the second image illustrated in the figure has different line lengths.

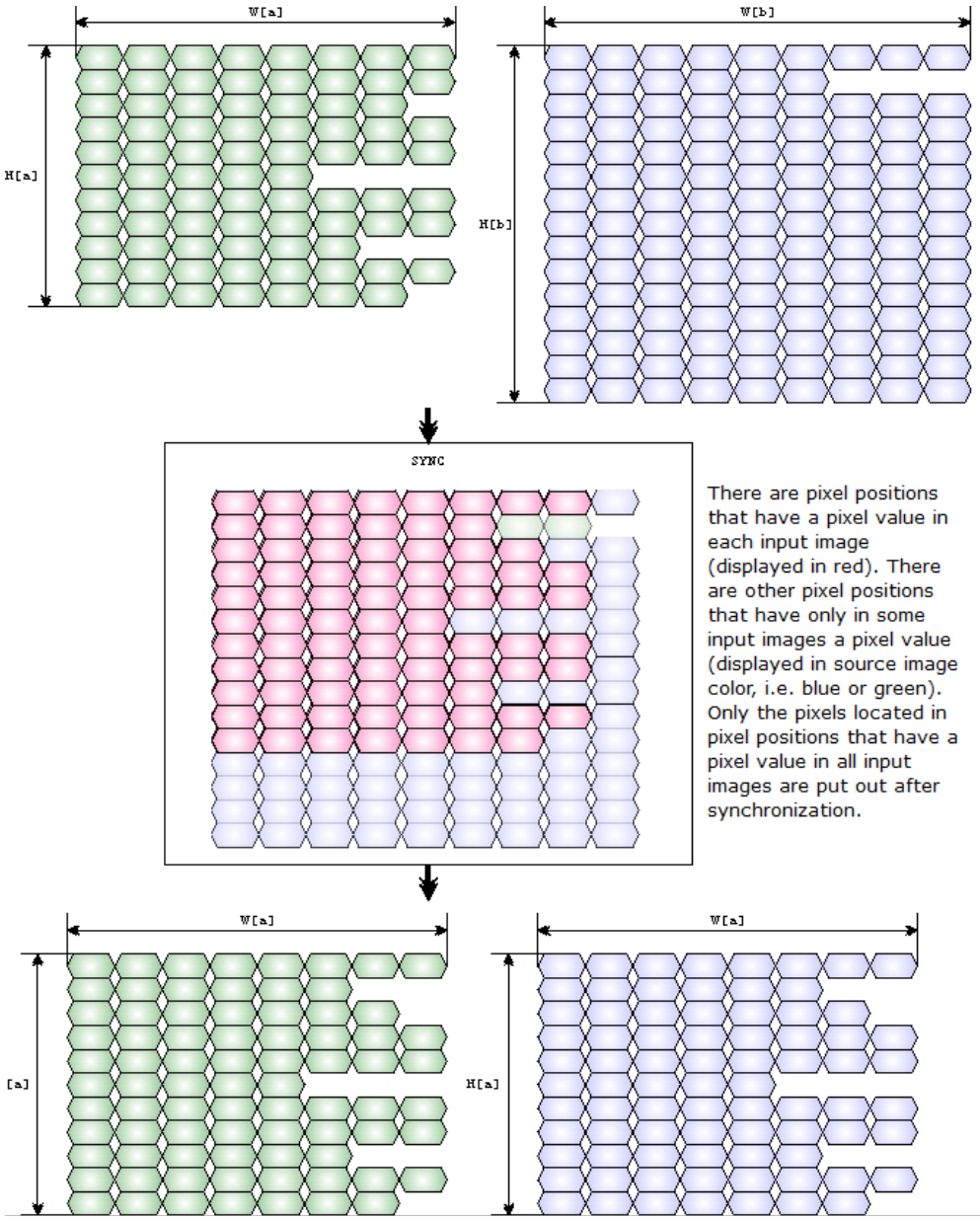
- 0D



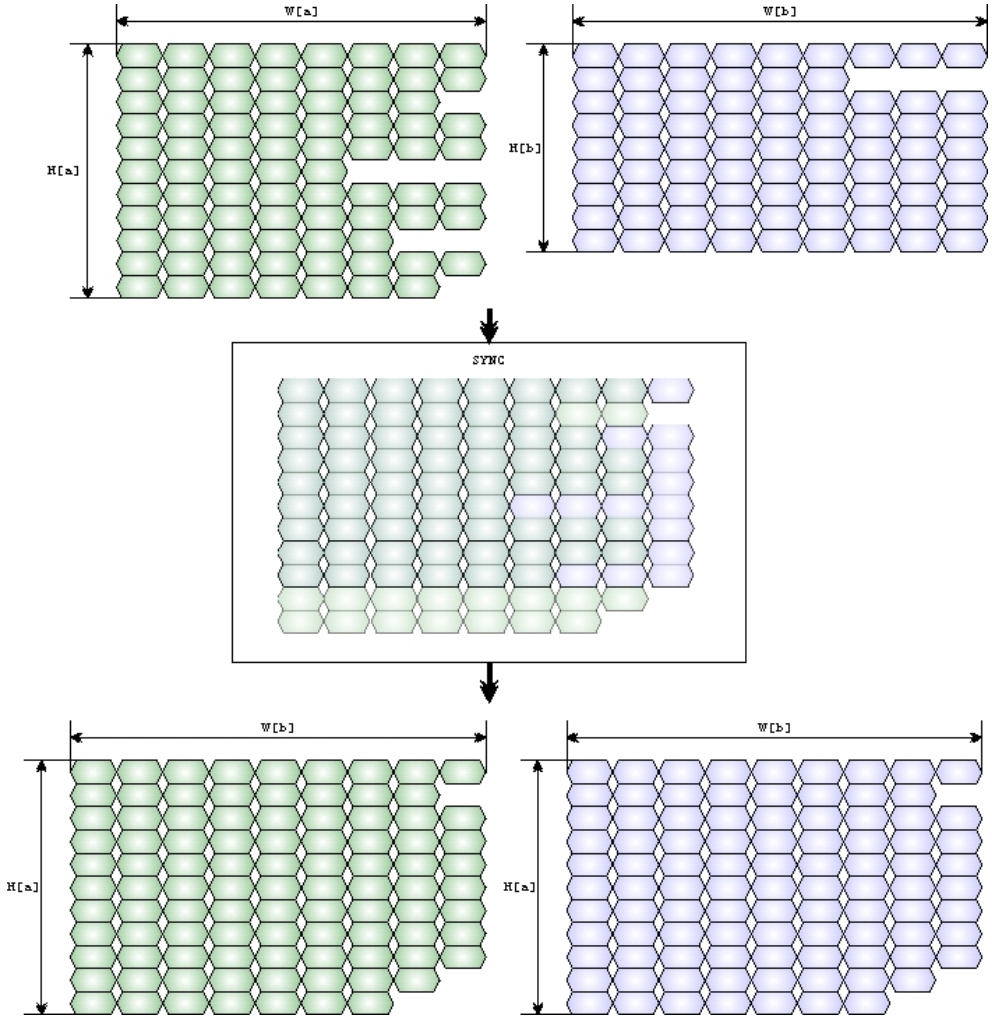
0D streams do not have any width or height. These streams are infinite data streams.

The image dimension synchronization is applied if the images on the input links have different image dimensions, i.e., a different width (or line length) and a different height. The image dimension synchronization guarantees that all images at the output are of the same dimension. The SYNC operator supports 2 synchronization modes: synchronization to the smallest image (SyncToMin) and synchronization to the largest image (SyncToMax). You select the synchronization mode via parameter *SyncMode*. The SyncToMin mode cuts larger images to fit into the smallest one. The SyncToMax mode expands small images to fit the largest one. The missing pixels are filled with dummy zero pixels (black pixels). Combining these 2 modes with the 3 different image domains, the following base synchronization combinations are possible:

### 32.26.2.1. 2D to 2D SyncToMin

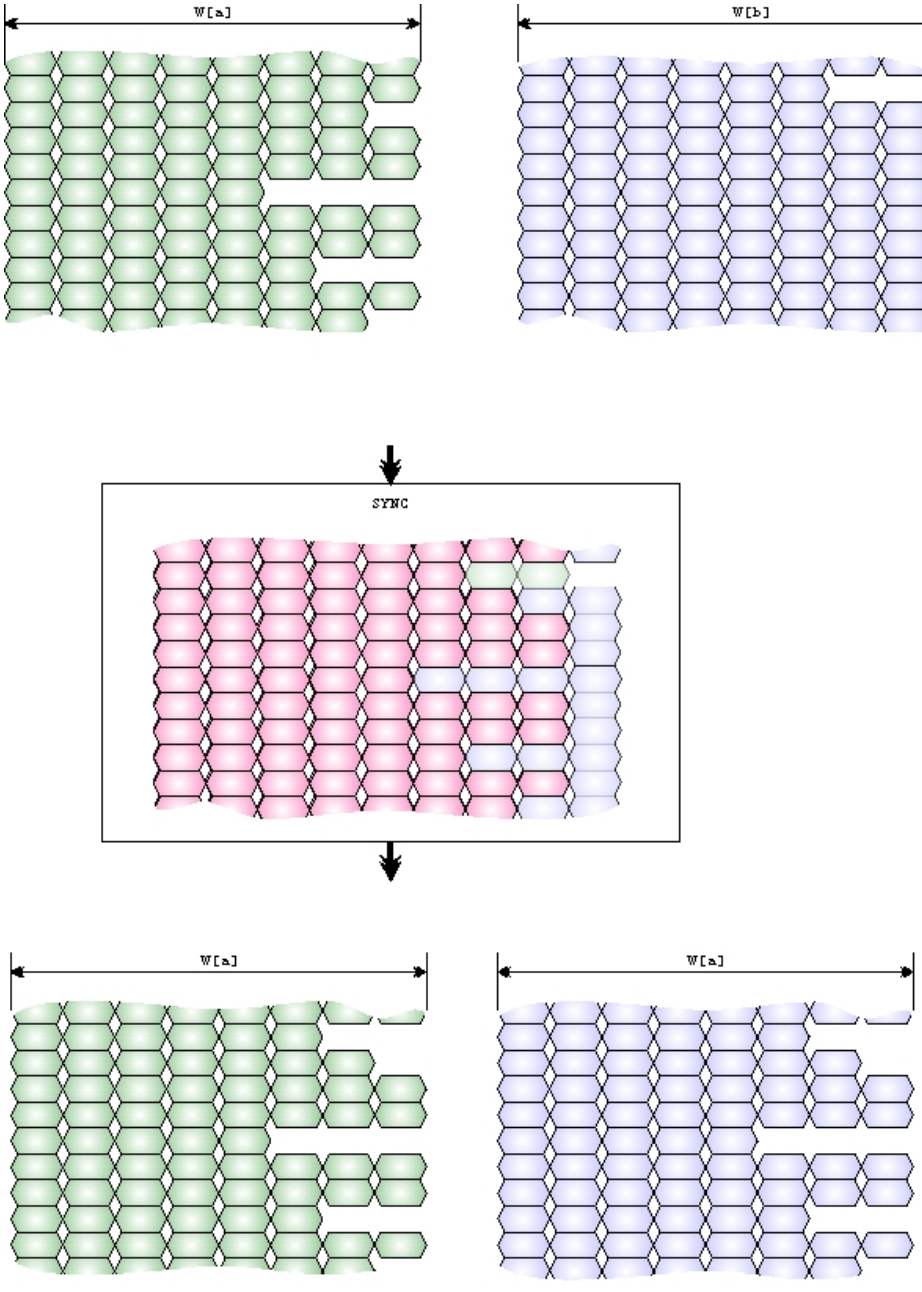


### 32.26.2.2. 2D to 2D SyncToMax

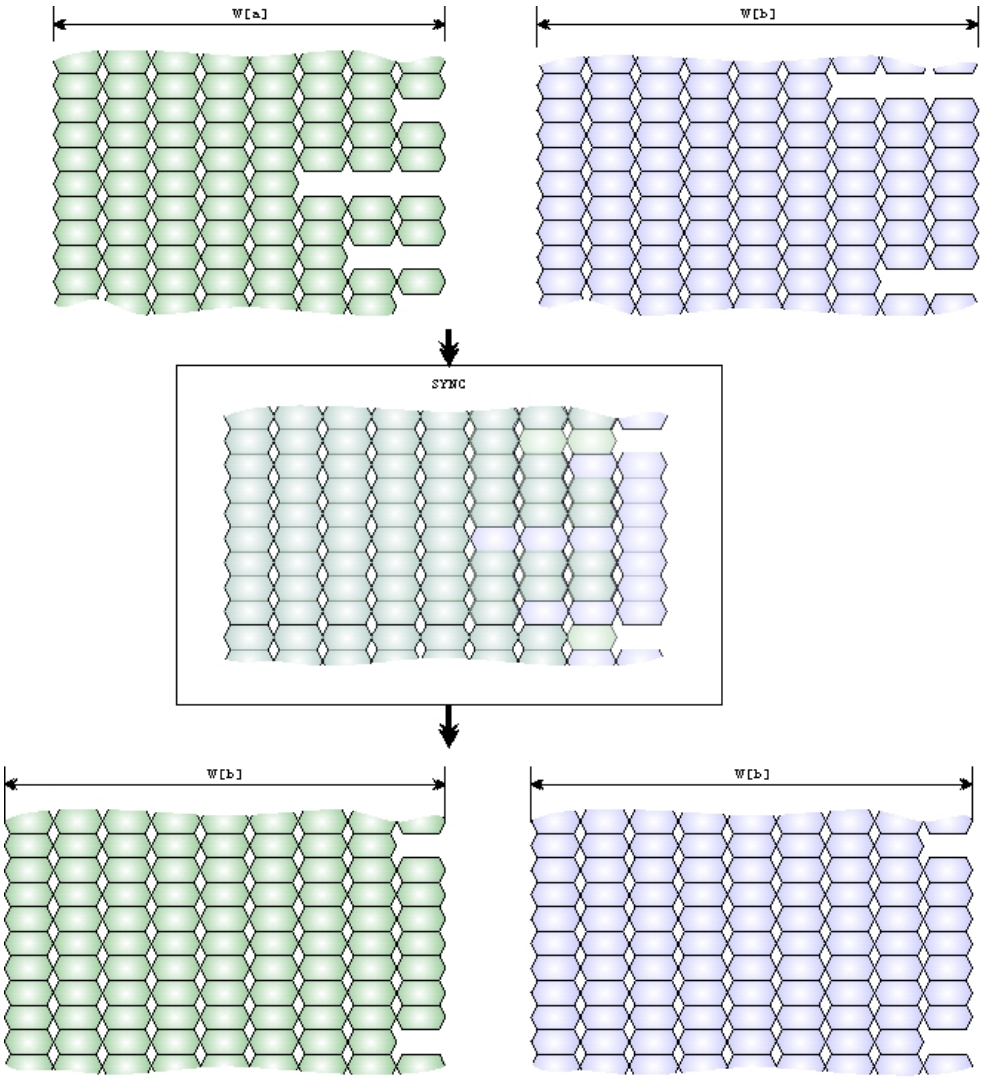



The padding pixels are zero pixels.

### 32.26.2.3. 1D to 1D SyncToMin



### 32.26.2.4. 1D to 1D SyncToMax

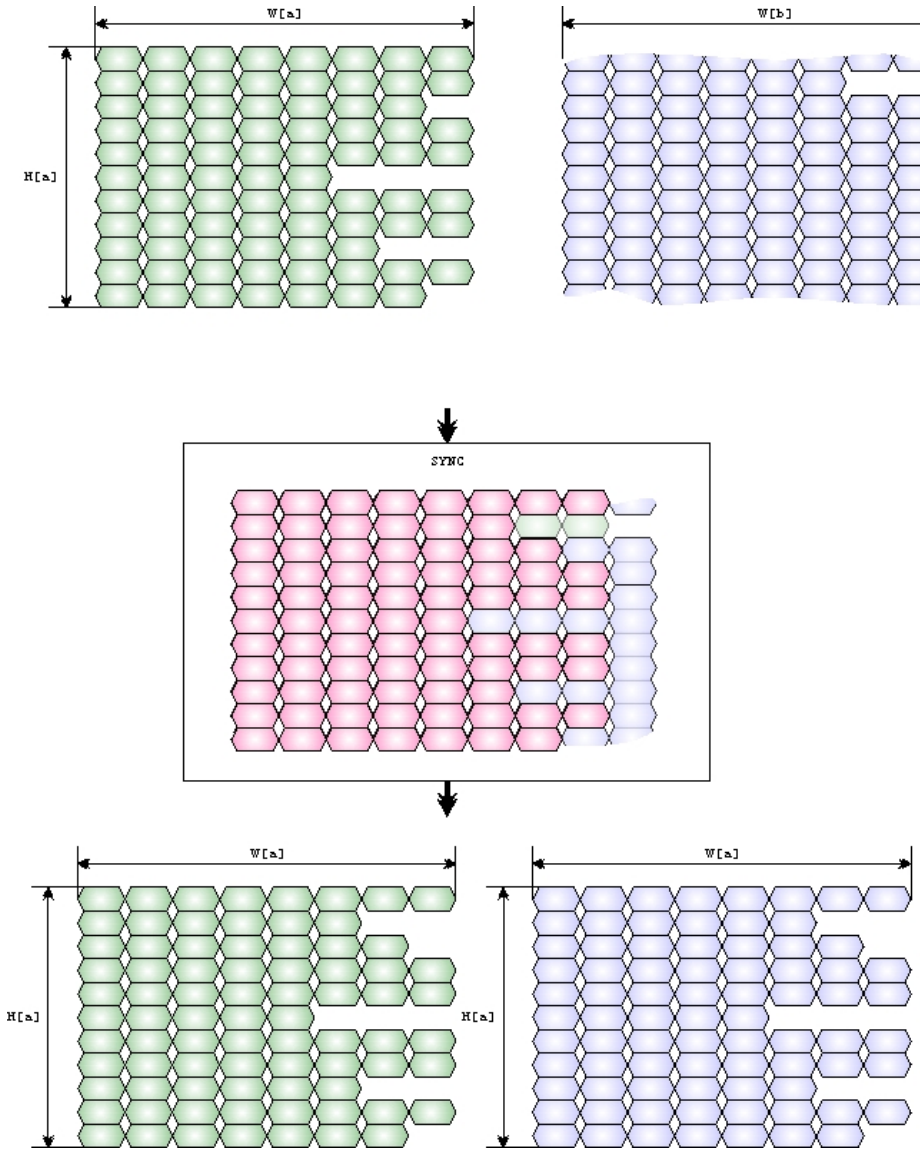



The padding pixels are zero pixels.

### 32.26.2.5. 0D to 0D SyncToMin / SyncToMax

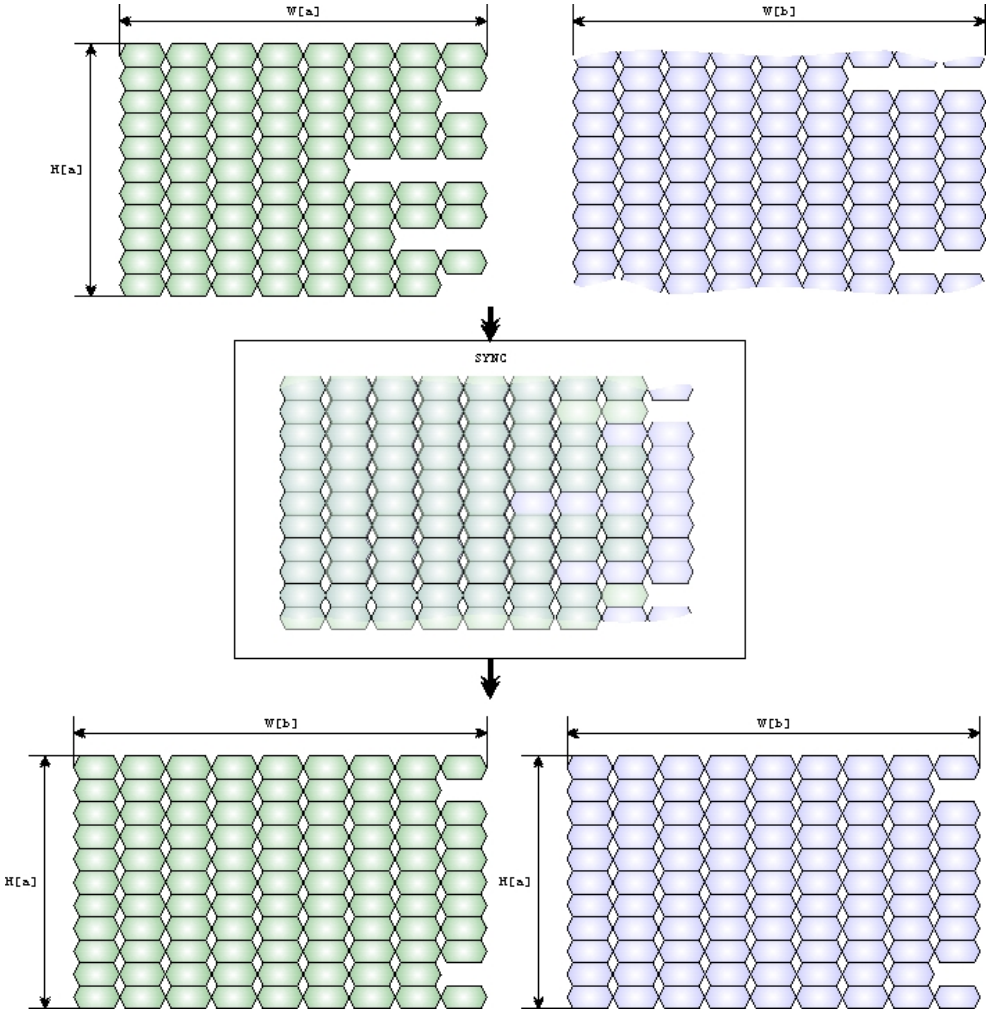
Synchronization between 0D images is a pure time synchronization since 0D images have no image dimensions.

### 32.26.2.6. 2D to 1D SyncToMin



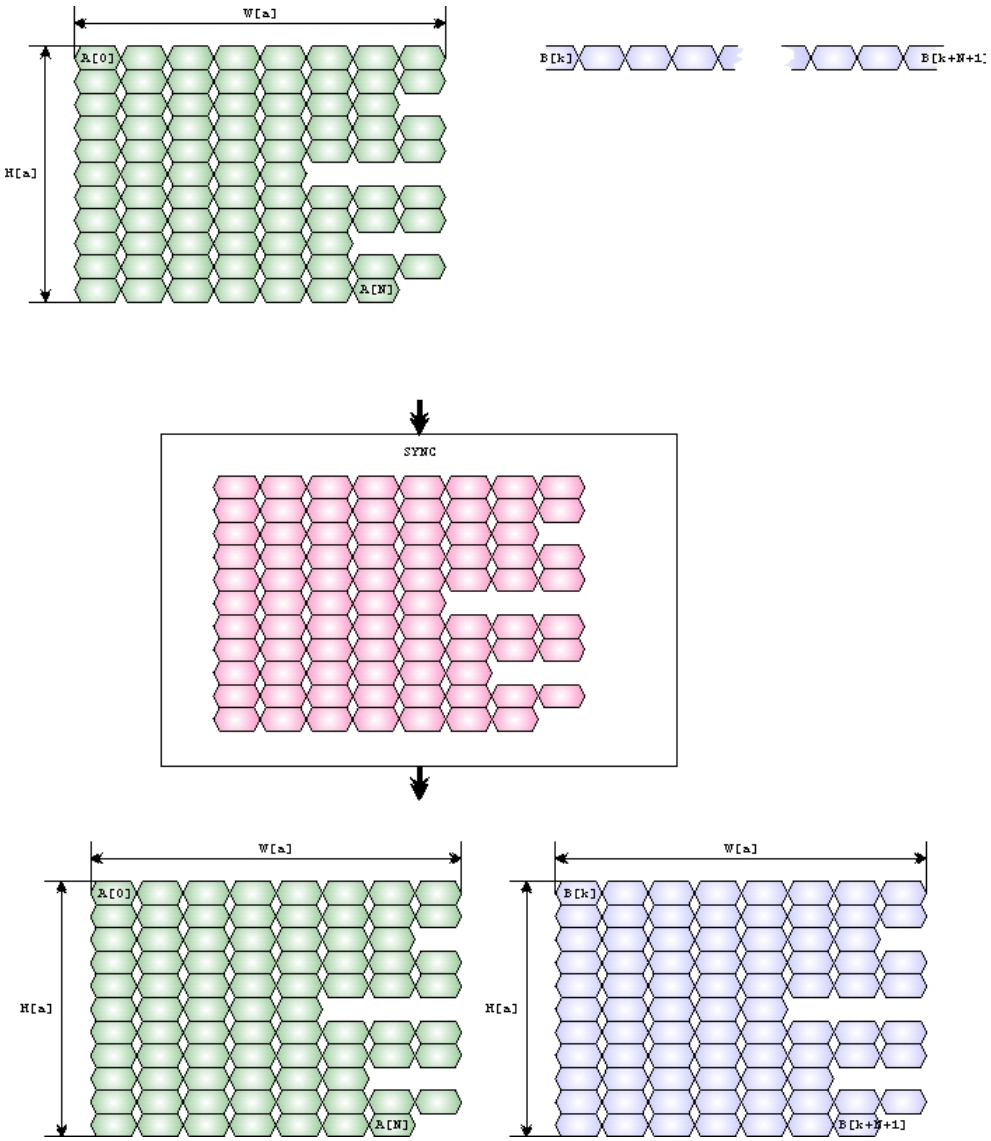

SyncToMin synchronization for 2D to 1D images performs an image dimension synchronization of the width. The output images keep the image height of the input 2D image. The 1D line stream is split into images which have the same height as the 2D image.

### 32.26.2.7. 2D to 1D SyncToMax



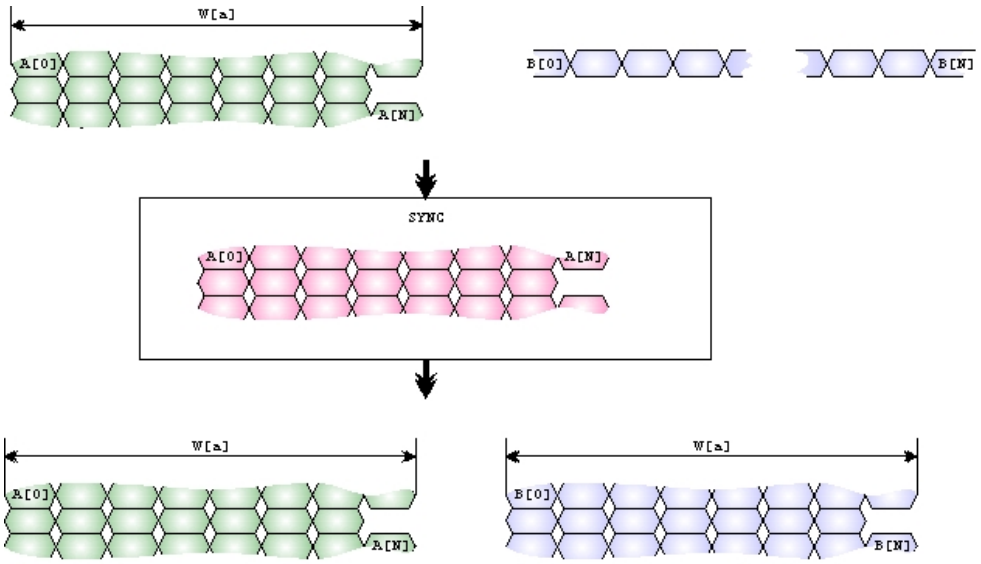

SyncToMax synchronization for 2D to 1D images performs an image dimension synchronization of the width. The output images keep the image height of the input 2D image. The 1D line stream is split into images which have the same height as the 2D image.

### 32.26.2.8. 2D to 0D SyncToMin / SyncToMax




Since 0D data streams have no shape the 2D to 0D synchronization is a simple bypass of both links. Of coures, the SYNC operator still performs the timing synchronization There is no difference between the SyncToMin and SyncToMax operation for this combination.

### 32.26.2.9. 1D to 0D SyncToMin / SyncToMax




Since 0D data streams have no shape the 1D to 0D synchronization is a simple bypass of both links. Of coures, the SYNC operator still performs the timing synchronization There is no difference between the SyncToMin and SyncToMax operation for this combination.

### 32.26.2.10. 2D to 1D to 0D SyncToMin / SyncToMax

This operation is a combination of 2D to 1D and 2D to 0D and 1D to 0D operations. The 2D image and the 1D line stream are synchronized in line lengths. The 1D link must provide at least the amount of lines the 2D image contains. Also the 0D link must provide at least the amount of pixels the result of 2D to 1D synchronization produces. If both of these conditions are fulfilled, the operator will output the result.



**Caution**

Warning: Simulation of mixed operaton modes like 2D to 1D or 1D to 0D or any combination of all 3 requires the following conditions to be met:

1. The links with inifinite height but finite width, i.e. 1D links, must contain at least the maximal (SyncToMax) / minimal (SyncToMin) amount of lines of all 2D channels participating in the synchronization.

2. The links with infinite width and height, i.e. 0D links, must provide at least the maximal (SyncToMax) / minimal (SyncToMin) amount of pixels the result of all 2D to 2D and 2D to 1D operations can produce.

3. Simulation of image sequences for mixed domain synchronization is not possible unless a special condition is met, because an infinite line cannot have a sequence on lines, otherwise it would not be infinite anymore. The same applies for the height. If the height is finite, it is not a 1D link but a 2D link. To simulate a sequence of images on all links, the amount of pixels on 0D channel must be exactly the same as the amount of pixels after 2D to 1D synchronization. The amount of lines on 1D channels must be exactly the same as the result after 2D to 1D synchronization. The exceeding lines (1D) respectively the exceeding pixels (0D) will not carry over to the next image sequence.

More information on the different image protocols can be found in Section 3.5.3, 'Image Protocols, Image Dimensions and Data Structure'.

## 32.26.3. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | I[0], data input<br>I[k], data input |
| **Output Link** | O[k], data output |

## 32.26.4. Supported Link Format

| Link Parameter | Input Link I[0] | Input Link I[k] | Output Link O[k] |
|---|---|---|---|
| **Bit Width** | [1, 64]❶ | [1, 64]❷ | as I[k] |
| **Arithmetic** | {unsigned, signed} | {unsigned, signed} | as I[k] |
| **Parallelism** | any | as I[0] | as I[0] |
| **Kernel Columns** | any | any | as I[k] |
| **Kernel Rows** | any | any | as I[k] |
| **Img Protocol** | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | {VALT_IMAGE2D,<br>VALT_LINE1D,<br>VALT_PIXEL0D} | auto❸ |
| **Color Format** | any | any | as I[k] |
| **Color Flavor** | any | any | as I[k] |
| **Max. Img Width** | any | any | auto❹ |
| **Max. Img Height** | any | any | auto❺ |

❶❷ The range of the input bit width is [1, 64] for unsigned inputs. For signed inputs, the range is [2, 64]. For unsigned color inputs, the range is [3, 63] and for signed color, the range is [6, 63].

❸ The output image protocol is VALT_IMAGE2D if at least one of the inputs is VALT_IMAGE2D otherwise VALT_LINE1D if at least one of the input is VALT_LINE1D. VALT_PIXEL0D of all inputs have image protocol VALT_PIXEL0D.

❹ If parameter *SyncMode* is SyncToMin the maximum ouput image width is the minimum of the input maximum image widths. If the parameter is set to SyncToMax the maximum ouput image width is the maximum of the input maximum image widths.

❺ If parameter *SyncMode* is SyncToMin the maximum ouput image height is the minimum of the input maximum image heights. If the parameter is set to SyncToMax the maximum ouput image height is the maximum of the input maximum image heights.

**Synchronous and Asynchronous Inputs**

- All inputs are asynchronous to each other i.e. they may be sourced by different M-type operators through an arbitrary network of O-type operators.

## 32.26.5. Parameters

| SyncMode | |
|---|---|
| **Type** | static parameter |
| **Default** | SyncToMin |
| **Range** | {SyncToMin, SyncToMax} |
| The parameter specifies the mode of operation: In SyncToMin mode the operator will synchronize all input images to the smalles image, i.e. cutting larger images. In SyncToMax mode the operator will expand all smaller images to the largest image. See descriptions above. | |

### 32.26.6. Examples of Use

The use of operator SYNC is shown in the following examples:

- Section 3.6.4, 'M-type Operators with Multiple Inputs'

  Synchronization Rules - Use of the SYNC Operator

- Section 3.6.6, 'Timing Synchronization'

  Synchronization - Avoiding deadlocks.

- Section 9.3.1.2, 'Combine Image Data From Two Camera Sources - Building an Overlay Blend'

  Tutorial - Synchronizing two cameras.

- Section 10.1.1.4, 'RGB Camera Link Medium Area'

  Tutorial - Basic Acquisition

- Section 10.1.1.5, 'Grayscale Camera Link Full Area'

  Tutorial - Basic Acquisition

- Section 11.8, 'Example for the *DMAFromPC* Operator on the imaFlex CXP-12 Quad Platform'

  Examples - Demonstration of how to use the operator using the example of shading correction

- Section 12.1.7, 'Laser Triangulation'

  Examples - A high speed and robust laser line detection algorithm. The algorithm determines center of gravity coordinates to obtain sub-pixel resolution results.

- Section 12.2.3, 'Histogram Threshold'

  Example - Histogram thresholding

- Section 12.4.1.7, 'Bayer De-Mosaicing Algorithm According to Laroche'

  Examples - Laroche Bayer De-Mosaicing filter

- Section 12.4.1.8, 'Modified Laroche Bayer De-Mosaicing Algorithm '

  Examples - Ressource Optimized Laroche Bayer De-Mosaicing filter

- Section 12.7.1, 'Motion Detection'

  Examples - Calculates the differences between two successive images. The differences are thresholded and output via DMA channel.

- Section 12.10.1, 'High Dynamic Range and Low Dynamic Range Example Using Camera Response Function'

  Examples - High Dynamic Range According to Debevec

- Section 12.10.2, 'High Dynamic Range and Low Dynamic Range Example with a Weighted Linear Ansatz'

  Examples - High Dynamic Range with Linear Ansatz

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

- Section 12.12.2, 'Depth From Focus Using Loops'

  Examples - Depth From Focus using Loops

- Section 12.14.3, '2D Shading Correction / Flat Field Correction'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in frame grabber RAM. The applet performs a high precision offset and gain correction.

- Section 12.14.4, '2D Shading Correction / Flat Field Correction Using Operator RamLUT'

  Examples - The example shows the implementation of a 2D shading correction. Correction values are stored in the operator *RamLUT*. The applet performs a high precision offset and gain correction.

# 32.27. Operator TxImageLink

**Operator Library: Synchronization**

The *TxImageLink* operator allows to send images to an *RxImageLink* operator any place in the design. Both operators establish a connection without a link.

With the image transfer between the *TxImageLink* and *RxImageLink* operators, it is possible to implement loops in a design.

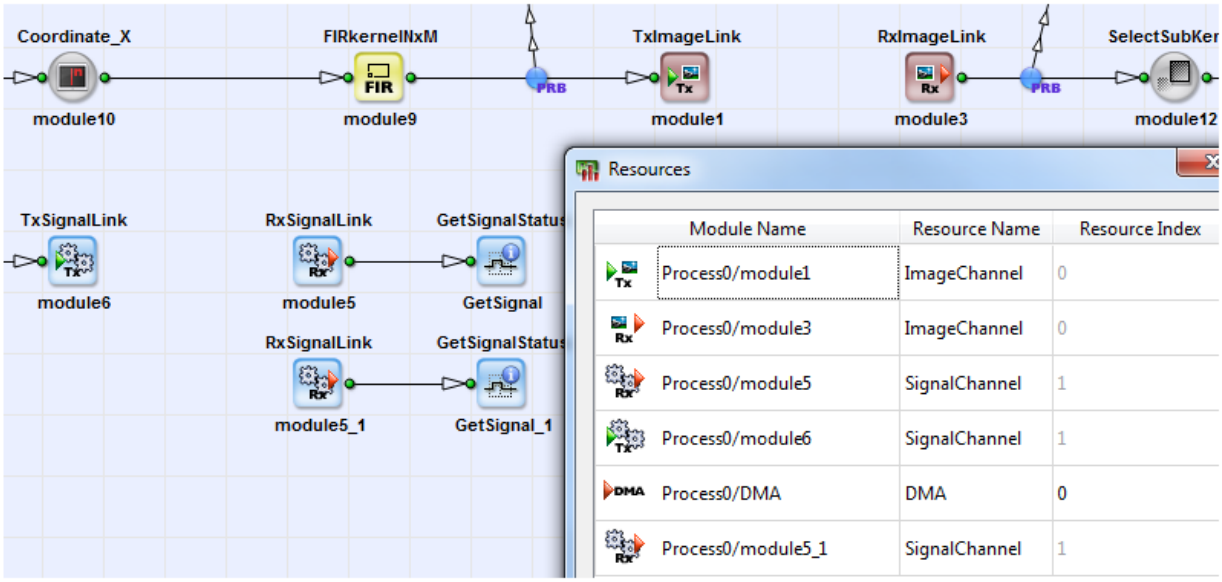
> **Loops require data buffering strategy**
>
> Operators *TxImageLink* and *RxImageLink* do not buffer data. Therefore, when implementing loops, you need to take special care with regard to data buffering to avoid deadlocks.

All image formats supported by VisualApplets in general are supported. The image format remains the same, i.e., the format at the ouput of RxImageLink is the same as the format *TxImageLink* receives at its input.

The parameter *Channel_ID* defines a channel ID to address the receiving *RxImageLink* operator. The parameter value has to be unique and must not be used by any other TxImageLink operator in the design.

There has to exist exactly one *RxImageLink* operator in the design which is using the same channel ID and receives the image data.

Each *TxImageLink* operator in a design is connected to exactly one *RxImageLink* operator via one channel ID. In the *Resource Dialog* of VisualApplets, you can see that one *ImageChannel* resource is used for each *TxImageLink-RxImageLink* connection. Resource *ImageChannel* allows to control the assignment of individual *TxImageLink* operators to individual *RxImageLink* operators. For the number of available *ImageChannel* resources (which also defines the maximum number of allowed *TxImageLink* and *RxImageLink* operators in a design), see Appendix A, '*Device Resources*'.

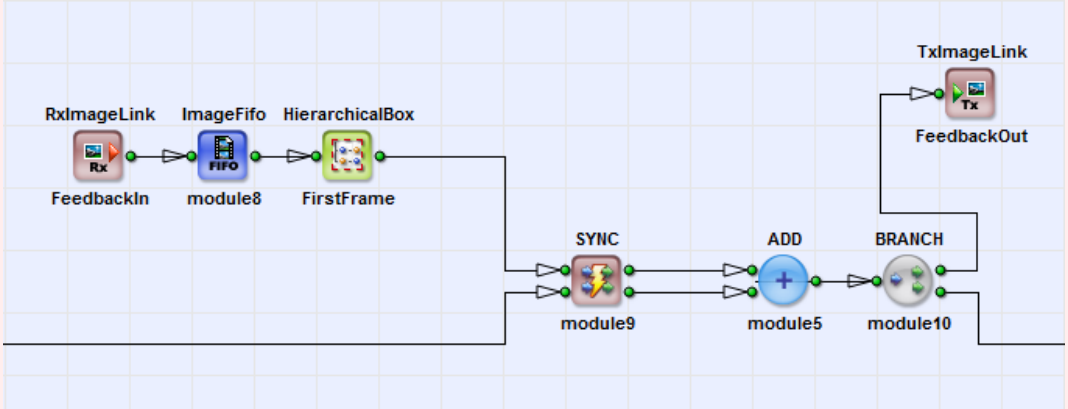



> **Parametrization of Link Format in Loops**
>
> As soon as the link properties dialog is opened, the "automatic update" feature will adapt the link properties (such as bit width, or image dimensions) according to the operator chain's configuration. The input format of *TxImageLink* always defines the

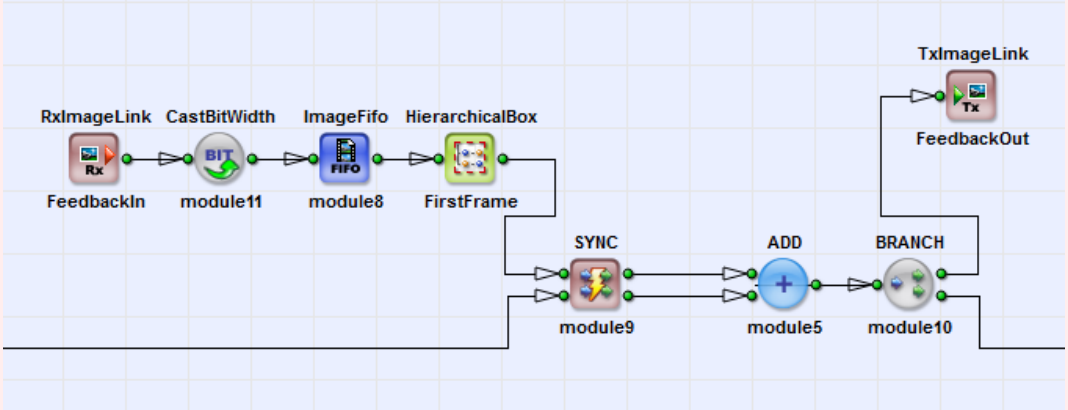
output format of *RxImageLink* (100% automatic consistency). Therefore, when you use operators *TxImageLink* and *RxImageLink* to implement loops, you need to take special care regarding the parametrization of the link formats.

**Example**

Wrong Implementation, value of link parameter *Bit Width* gets higher with every iteration through the loop:




Right Implementation, Operator CastBitWidth changes the value of link parameter *Bit Width* back to the original value with every iteration through the loop:




**Synchronizing Channels between different hierarchical design levels**

If you use a TxImageLink/RxImageLink pair to set up a data transfer channel between different hierarchical design levels, this connection is treated as an M operator. Therefore, you may need to implement additional synchronization elements (that are not required with a direct connection).

## 32.27.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, data input |
| **Output Link** | O, data output |

### 32.27.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64]❶ | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |
| **Kernel Rows** | any | as I |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D, VALT_PIXEL0D, VALT_SIGNAL} | as I |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

❶ The range of the input bit width is [1, 64] for unsigned values. For signed inputs, the range is [2, 64]. For unsigned color inputs [3, 63] and for signed color inputs [6, 63].

### 32.27.3. Parameters

| **Channel_ID** | |
|---|---|
| **Type** | static parameter |
| **Default** | 0 |
| **Range** | [0, 1023] |
| The channel ID of the image link. See descriptions above. | |

### 32.27.4. Examples of Use

The use of operator TxImageLink is shown in the following examples:

- Section 11.4, 'Functional Example for Loading Test Images Using *ImageInjector* '

  Examples - Demonstration of how to use the operator

- Section 12.12.1, 'A rolling average is applied on a dynamic number of images'

  Examples - Rolling Average - Loop

- Section 12.12.2, 'Depth From Focus Using Loops'

  Examples - Depth From Focus using Loops

# 32.28. Operator Overflow

**Operator Library: Synchronization**

The operator *Overflow* decouples a non-stoppable ingoing 2D data stream from a subsequent image processing pipeline which may be blocked sometimes. It identifies an overflow situation when an internal small buffer runs full as outgoing data is blocked. In case of an overflow, an ongoing frame is cut by immediately appending an end-of-line and end-of-frame marker. The corresponding frame is considered as corrupted and an event is generated accordingly. When new frames enter input I while blocking of the output remains full, frames are skipped. Then an event about a lost frame is generated.



**Overflow Causes Non-rectangular Frames**

When an ingoing frame is cut because of an overflow situation, the last line of the outgoing frame is typically shorter than the other lines. Make sure that the subsequent image processing pipeline can deal with that.

For each corrupted or lost frame an overflow event is generated. The overflow events transport a data payload consisting of the frame number and the type of overflow. The event payload is provided as three 16-bit data words. The first two represent a 32-bit value for the frame number where the least significant bits are stored in the first word. The third 16-bit word provides an overflow mask where the mask bits are defined as follows:

- [0]: frame corrupted
- [1]: frame lost
- [2]: event loss occurred before
- [3]: frame ok
- [4] .. [15]: reserved

Note that the frame number is reset on acquisition start. Also note that the first frame will have frame number zero, while a DMA transfer starts with frame number one. The frame number is a 32-bit value. If it's maximum is reached, it will start from zero again. On a 64-bit target runtime, the DMA transfer number will be a 64-bit value. If the *frame corrupted* is set, the frame with the frame number in the event is corrupted i.e. it will not have it's full length so subsequent processing may produce invalid results. If the *frame lost* flag is set, the frame with the frame number in the event was fully discarded so the subsequent processing pipeline did not receive any data from the frame. The *corrupted frame* flag and the *frame lost* flag will never occur for the same event. The flag *event loss occured before* is an additional security mechanism. It means that an event has been lost. This can only happen at very high event rates and should not happen under normal conditions.

## 32.28.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Link** | I, non-stoppable data input |
| **Output Link** | O, data output |

## 32.28.2. Supported Link Format

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Bit Width** | [1, 64] | as I |
| **Arithmetic** | {unsigned, signed} | as I |
| **Parallelism** | any | as I |
| **Kernel Columns** | any | as I |

| Link Parameter | Input Link I | Output Link O |
|---|---|---|
| **Kernel Rows** | any | as I |
| **Img Protocol** | VALT_IMAGE2D | VALT_IMAGE2D |
| **Color Format** | any | as I |
| **Color Flavor** | any | as I |
| **Max. Img Width** | any | as I |
| **Max. Img Height** | any | as I |

## 32.28.3. Parameters

| **EventsForSuccessfulFrame** | |
|---|---|
| **Type** | dynamic read/write parameter |
| **Default** | OFF |
| **Range** | [OFF, ON] |
| Defines whether an event for any passing frame shall be generated. | |

| **OverflowOccurred** | |
|---|---|
| **Type** | dynamic read parameter |
| **Default** | 0 |
| **Range** | [0, 1] |
| Notifies if an overflow event occurred.<br><br>This parameter is reset after read access so it shows whether an overflow situation occurred between the prevous read access and the current read access. | |

## 32.28.4. Examples of Use

The use of operator Overflow is shown in the following examples:

- Section 13.6, 'Functional Example for Specific Operators of Library Synchronization, Base and Filter'

  Examples - Demonstration of how to use the operator

# 33. Library Transformation




The *Transformation* library include operators for image transformation like FFT.

The following list summarizes all Operators of Library Transformation

| Operator Name | | Short Description | available since |
|---|---|---|---|
|  | FFT | One dimensional Fast Fourier transform (FFT) | Version 2.2.0 |

Table 33.1. Operators of Library Transformation

# 33.1. Operator FFT

**Operator Library: Transformation**

Operator FFT performs a forward or reverse one dimensional Fast Fourier transform. The operator accepts complex values at the input and will output complex values. It uses two ports at the input and output. One for the real part and another one for the imaginary part.

By using parameters, the operator can be configured to the number of points i.e. line width. The line width has to match with the number of pixel at the input.

A second parameter defines if the operator performs a forward or reverse transformation.



**Limited Configurations Only**

The operator currently can only be used on imaFlex CXP-12 Penta, imaFlex CXP-12 Quad, microEnable 5 ironman and microEnable 5 marathon and LightBridge frame grabbers in specific configuration.

The operator is limited to imaFlex CXP-12 Penta, imaFlex CXP-12 Quad, mE5 ironman and mE5 matathon/LightBridge. In the following the allowed configurations and characteristics are listed.

**imaFlex CXP-12 Penta, and imaFlex CXP-12 Quad:**

- **Points (image width):** {8, 16, 32, 64, 128, 256, 512, 1024, 2048}
- **Parallelism:** 1
- **Input bit width:** 32
- **Output bit width:** 48

**mE5 ironman:**

- **Points (image width):** {8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192}
- **Parallelism:** 2
- **Input bit width:** 16
- **Output bit width:** 32

**mE5 marathon / LightBridge:**

- **Points (image width):** {8, 16, 32, 64, 128, 256, 512, 1024, 2048}
- **Parallelism:** 2
- **Input bit width:** 32
- **Output bit width:** 48

**The operator uses the XILINX FFT IP core with following parameterization:**

- **Architecture:** Pipelined Streaming IO
- **Run Time Configurable Transform Length:** yes
- **Data Format:** Fixed Point
- **Twiddle Width (Phase Factor Width):** 10
- **Scaling Option:** Unscaled
- **Rounding Mode:** truncation

More information about the FFT algorithm can be found at https://www.xilinx.com/products/intellectual-property/fft.html#overview. imaFlex CXP-12 Penta and imaFlex CXP-12 Quad use Xilinx IP

version 9.1 [https://docs.xilinx.com/r/en-US/pg109-xfft]. marathon and ironman use Xilinx IP version 8.0 [https://docs.xilinx.com/v/u/en-US/pg003_v_enhance].

### 33.1.1. I/O Properties

| Property | Value |
|---|---|
| **Operator Type** | M |
| **Input Links** | IRe, input real part<br>IIm, input imaginary part |
| **Output Links** | ORe, output real part<br>OIm, output imaginary part |

**Synchronous and Asynchronous Inputs**

- All inputs are synchronous to each other i.e. they have to be sourced by the same M-type operator through an arbitrary network of O-type operators.

### 33.1.2. Supported Link Format

| Link Parameter | Input Link IRe | Input Link IIm |
|---|---|---|
| **Bit Width** | 16 or 32 ❶ | as IRe |
| **Arithmetic** | signed | signed |
| **Parallelism** | depends on platform❷ | depends on platform❷ |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | {VALT_IMAGE2D, VALT_LINE1D} | as IRe |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | depends on platform❸ | as IRe |
| **Max. Img Height** | any | as IRe |

| Link Parameter | Output Link ORe | Output Link OIm |
|---|---|---|
| **Bit Width** | auto❹ | as ORe |
| **Arithmetic** | signed | signed |
| **Parallelism** | depends on platform❷ | depends on platform❷ |
| **Kernel Columns** | 1 | 1 |
| **Kernel Rows** | 1 | 1 |
| **Img Protocol** | as IRe | as IRe |
| **Color Format** | VAF_GRAY | VAF_GRAY |
| **Color Flavor** | FL_NONE | FL_NONE |
| **Max. Img Width** | as IRe | as IRe |
| **Max. Img Height** | as IRe | as IRe |

❶ The input bit width is 16 bit on mE5 ironman and 32 bit in mE5 marathon/LightBridge.

❹ The output bit width is 32 bit on mE5 ironman and 64 bit in mE5 marathon/LightBridge.

❷ The parallelism is 1 for imaFlex CXP-12 Penta and imaFlex CXP-12 Quad, and 2 for mE5 ironman as well as mE5 marathon/LightBridge.

❸

[2^3, 2^13] on mE5 ironman, [2^3, 2^11] on mE5 marathon/LightBridge.

### 33.1.3. Parameters

| TransformationMode | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Forward |
| **Range** | {Forward, Reverse} |
| Specify a forward or reverse transform. | |

| LineLength | |
|---|---|
| **Type** | dynamic write parameter |
| **Default** | Points1024 |
| **Range** | {8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192} on mE5 ironman, {8, 16, 32, 64, 128, 256, 512, 1024, 2048} on mE5 marathon/LightBridge and imaFlex CXP-12 Penta and imaFlex CXP-12 Quad |
| Specify the number of transformation points i.e. the line length. | |

### 33.1.4. Examples of Use

The use of operator FFT is shown in the following examples:

- Section 12.1.1, 'Fast Fourier transform'

  Shows the usage of operator *FFT*.

# Appendix A. Device Resources

The following lists show important hardware details of all supported hardware platforms of this VisualApplets version. For a detailed list, please check the data sheet of the individual product.

## A.1. Hardware Configuration of Supported Platforms

### A.1.1. microEnable IV and PixelPlant

| Resource | mE4VD1-CL/-PoCL | mE4VD4-CL/-PoCL | mE4VQ4-GE/-GPoE | px100 | px200/ px200e |
|---|---|---|---|---|---|
| Vision Processor | Xilinx Spartan 3 XC3S1600e FPGA | Xilinx Spartan 3 XC3S4000 FPGA | Xilinx Spartan 3 XC3S4000 FPGA | Xilinx Spartan 3 XC3S1600e FPGA | Xilinx Spartan 3 XC3S4000 FPGA |
| LUT | 29504 | 55296 | 55296 | 29504 | 55296 |
| Flip-Flop | 29504 | 55296 | 55296 | 29504 | 55296 |
| Block RAM* | 36 à 18432Bit | 96 à 18432Bit | 96 à 18432Bit | 36 à 18432Bit | 96 à 18432Bit |
| Embedded Arithmetic Logic Unit (ALU)* | 36 | 96 | 96 | 36 | 96 |
| RAM | 2 x 128MiB DDR | 4 x 128MiB DDR | 4 x 128MiB DDR | 2 x 128MiB DDR | 4 x 128MiB DDR |
| Data Width per RAM | 64Bit | 64Bit | 64Bit | 64Bit | 64Bit |
| Bandwidth per RAM | 1GB/s | 1GB/s | 1GB/s | 1GB/s | 1GB/s |
| Base Design Clock | 62.5MHz | 62.5MHz | 62.5MHz | 62.5MHz | 62.5MHz |
| Host Interface | PCIe x1 | PCIe x4 | PCIe x4 | - | - |
| Host Interface (PCIe x 4 Gen 2) Bandwidth (theor.) | 250 Mbyte/s per direction on PCIe bus | 1 Gbyte/s per direction on PCIe bus | 1 Gbyte/s per direction on PCIe bus | - | - |
| Host Interface (PCIe x 4 Gen 2) Bandwidth (typ./max.) | 200 Mbyte/s | 750 Mbyte/s / 900 MByte/s | 750 Mbyte/s | - | - |
| * microEnable IV and PixelPlant only: Block RAM and hardware multiplier are shared. The given value is the total value. | | | | | |

Table A.1. Hardware Configuration microEnable IV and PixelPlant

### A.1.2. microEnable 5 ironman

| Resource | mE5VD8-PoCL | mE5VQ8-CXP6B/mE5VQ8-CXP6D |
|---|---|---|
| Vision Processor | Xilinx Virtex6 XC6VLX240T FPGA | Xilinx Virtex6 XC6VLX240T FPGA |
| LUT | 150720 | 150720 |

| Resource | mE5VD8-PoCL | mE5VQ8-CXP6B/mE5VQ8-CXP6D |
|---|---|---|
| Flip-Flop | 301440 | 301440 |
| Block RAM | 832 x 18432Bit | 832 x 18432Bit |
| Embedded Arithmetic Logic Unit (DSP48) | 768 | 768 |
| RAM | 4 x 256MiB DDR3 | 4 x 256MiB DDR3 |
| Data Width per RAM | 128Bit | 128Bit |
| Bandwidth per RAM | 4GB/s | 4GB/s |
| Base Design Clock | 125MHz | 125MHz |
| Host Interface | PCIe x8 Gen2 | PCIe x8 Gen2 |
| Host Interface (PCIe x 8 Gen 2) Bandwidth (theor.) | 4 Gbyte/s per direction on PCIe bus | 4 Gbyte/s per direction on PCIe bus |
| Host Interface (PCIe x 8 Gen 2) Bandwidth (typ./max.) | up to 3.6 GByte/s on PCIe bus | up to 3.6 GByte/s on PCIe bus |

Table A.2. Hardware Configuration microEnable 5 ironman

### A.1.3. LightBridge and microEnable 5 marathon

| Resource | mE5 marathon VCX-QP | mE5 marathon VF2 | mE5 marathon VCL | mE5 marathon VCLx | LightBridge 2 VCL |
|---|---|---|---|---|---|
| Vision Processor | Xilinx Kintex7 XC7K160T - 2FFG676C FPGA | Xilinx Kintex7 XC7K160T - 2FFG676C FPGA | Xilinx Kintex7 XC7K160T - 1FBG676C FPGA | Xilinx Kintex7 XC7K410T - 1FBG676C FPGA | Xilinx Kintex7 XC7K160T - 1FBG676C FPGA |
| LUT | 101400 | 101400 | 101400 | 254200 | 101400 |
| Flip-Flop | 202800 | 202800 | 202800 | 508400 | 202800 |
| Block RAM (18k) | 650 | 650 | 650 | 1590 | 650 |
| Embedded Arithmetic Logic Unit (DSP48) | 600 | 600 | 600 | 1540 | 600 |
| RAM size | 4 x 512MiB DDR3 | 4 x 512MiB DDR3 | 4 x 512MiB DDR3 | 4 x 512MiB DDR3 | 4 x 512MiB DDR3 |
| RAM Data Width | 512 Bit | 512 Bit | 256 Bit | 256 Bit | 256 Bit |
| RAM Bandwidth total (shared) | 12.8 GB/s* | 12.8 GB/s* | 6.4 GB/s* | 6.4 GB/s* | 6.4 GB/s* |
| Base Design Clock (default) | 125MHz | 125MHz | 125MHz | 125MHz | 125MHz |
| Base Design Clock (maximal) | 312.5 MHz** | 312.5 MHz** | 312.5 MHz** | 312.5 MHz** | 312.5 MHz** |
| Host Interface | PCIe x 4 Gen 2 (Direct Memory Access) | PCIe x 4 Gen 2 (Direct Memory Access) | PCIe x 4 Gen 2 (Direct Memory Access) | PCIe x 4 Gen 2 (Direct Memory Access) | PCIe x 4 Gen 2 interface via Thunderbolt™ 2 technology |
| Host Interface (PCIe x 4 Gen | 1x2000 MB/s | 1x2000 MB/s | 1x2000 MB/s | 1x2000 MB/s | 1x2000 MB/s |

| Resource | mE5 marathon VCX-QP | mE5 marathon VF2 | mE5 marathon VCL | mE5 marathon VCLx | LightBridge 2 VCL |
|---|---|---|---|---|---|
| 2) Bandwidth (theor.) | | | | | |
| Host Interface (PCIe x 4 Gen 2) Bandwidth (typ./max.) | Up 1800 MB/s sustainable data bandwidth | Up to 1800 MB/s sustainable data bandwidth | Up to 1800 MB/s sustainable data bandwidth | Up to 1800 MB/s sustainable data bandwidth | Up to 1400 MB/s sustainable data bandwidth |
| * The platforms own only one single physical RAM bank which is formatted as 4 independent, non-overlapping memory regions. Though the memory itself is exclusive for each RAM based operator, the RAM bandwidth is shared. See section Shared Memory Concept.<br><br>** These platforms allow to use a user-specified base clock. The minimum clock frequency is 125 MHz. Theoretical maximum is 312.5 MHz. Designs with a clock frequency of 125 MHz are likely to meet the timing constraints. Designs with a clock frequency above 125 MHz may result in timing constraints violations. | | | | | |

Table A.3. Hardware Configuration LightBridge and microEnable 5 marathon

### A.1.4. imaFlex CXP-12 Quad and imaFlex CXP-12 Penta

| Resource | imaFlex CXP-12 Quad | imaFlex CXP-12 Penta |
|---|---|---|
| Vision Processor | Xilinx UltraScale+ XCKU3P-FFVD900-1-E | Xilinx UltraScale+ XCKU3P-FFVB676-1-E |
| LUT | 160679 | 161049 |
| Flip-Flop | 323224 | 323216 |
| Block RAM (18k) | 720 | 720 |
| Embedded Arithmetic Logic Unit (DSP48) | 1368 | 1368 |
| RAM size | 3 x 512 MiB DDR4 | 5 x 512 MiB DDR4 |
| RAM Data Width | 384 Bit | 640 Bit |
| RAM Bandwidth total (shared) | 14.4 GB/s* | 24.0 GB/s* |
| Base Design Clock (default) | 312.5 MHz** | 312.5 MHz** |
| Base Design Clock (maximal) | 400.0 MHz | 400.0 MHz |
| Host Interface | PCIe x 8 Gen 3 (Direct Memory Access) | PCIe x 8 Gen 3 (Direct Memory Access) |
| Host Interface (PCIe x 8 Gen 3) Bandwidth (theor.) | 8000 MB/s | 8000 MB/s |
| Host Interface (PCIe x 8 Gen 3) Bandwidth (typ./max.) | 7200 MB/s sustainable data bandwidth | 7200 MB/s sustainable data bandwidth |
| * The platform owns only one single physical RAM bank. Though the memory itself is exclusive for each RAM based operator, the RAM bandwidth and RAM size is shared. See section Section A.3, ' Shared Memory Concept '.<br><br>** This platform allows to use a user-specified base clock. The minimum clock frequency is 312.5 MHz. Theoretical maximum is 400 MHz. Designs with a clock frequency of 312.5 MHz are likely to meet the timing constraints. Designs with a clock frequency above 312.5 MHz may result in timing constraints violations, dependent on the applet algorithms implementation. | | |

Table A.4. Hardware Configuration imaFlex CXP-12 Quad and imaFlex CXP-12 Penta

## A.2. Device Resources of Supported Platforms

The device resources are limited for each hardware device. The following lists show the available device resources of the supported hardware platforms. Operators use device resources. Each resource can only be used once.

Device resources are allocated either

- automatically,
- using operator parameters, or
- in the *Resources* dialog.

See Section 3.8, 'Allocation of Device Resources' for more information.

### A.2.1. microEnable IV and PixelPlant

| Resource | mE4VD1-CL/-PoCL | mE4VD4-CL/-PoCL | mE4VQ4-GE/-GPoE | px100 | px200/ px200e |
|---|---|---|---|---|---|
| CAM | 2 | 2 | 4 | | |
| CameraControl | 2 | 2 | | | |
| DMA | 8 | 8 | 8 | | |
| Event | | 64 | 64 | | |
| EventSource | | 14 | 14 | | |
| RAM | 2 x 128MiB | 4 x 128MiB | 4 x 128MiB | 2 x 128MiB | 4 x 128MiB |
| RxLink | 60 | 60 | 60 | 60 | 60 |
| TriggerOut* | 8 | 8 | 8 | 8 | 8 |
| TxLink | 60 | 60 | 60 | 60 | 60 |
| SignalChannel** | 4000 | 4000 | 4000 | 4000 | 4000 |
| ImageChannel*** | 1024 | 1024 | 1024 | 1024 | 1024 |

* This resource is not visible in the Resources dialog. It is controlled via operators.

** Resource *SignalChannel* allows to connect *TxSignalLink* operators with *RxSignalLink* operators. Each operator *TxSignalLink* needs one resource *SignalChannel* exclusively. Multiple operators *RxSignalLink* can use the same resource *SignalChannel*, i.e., multiple operators *RxSignalLink* can receive the signals transmitted by one operator *TxSignalLink*. A maximum of 4000 *TxSignalLink* operators can be used in a design. The number of *RxSignalLink* operators is not restricted. Resource *SignalChannel* is visible in the *Resources* dialog.

*** Resource *ImageChannel* allows to connect *TxImageLink* operators with *RxImageLink* operators. Each operator *TxImageLink* needs one resource *ImageChannel* exclusively. Each resource *ImageChannel* can be connected to exactly one operator *RxImageLink*, i.e., a maximum of 1024 *TxImageLink* and 1024 *RxImageLink* operators can be used in one design. Resource *ImageChannel* is visible in the *Resources* dialog.

Table A.5. List of Device Resources microEnable IV and PixelPlant

### A.2.2. microEnable 5 ironman

| Resource | mE5VD8-PoCL | mE5VQ8-CXP6D/mE5VQ8-CXP6B |
|---|---|---|
| Camera Port | 2 | 4 |

| Resource | mE5VD8-PoCL | mE5VQ8-CXP6D/mE5VQ8-CXP6B |
|---|---|---|
| DMARd | 4 | 4 |
| GPO* | 8 OUT | 8 OUT |
| RAM | 4 x 256MiB | 4 x 256MiB |
| LED Ports* | 4 | 4 |
| GPI** | 8 | 8 |
| SignalChannel*** | 4000 | 4000 |
| EventPort**** | 12 | 14 |
| EventID**** | 64 | 64 |
| ImageChannel***** | 1024 | 1024 |

* These resources are not visible in the *Resources* dialog. They are controlled via operators.

** GPI is not visible in the *Resources* dialog. The same resource can be used multiple times. The table lists the amount of GPI ports.

*** Resource *SignalChannel* allows to connect *TxSignalLink* operators with *RxSignalLink* operators. Each operator *TxSignalLink* needs one resource *SignalChannel* exclusively. Multiple operators *RxSignalLink* can use the same resource *SignalChannel*, i.e., multiple operators *RxSignalLink* can receive the signals transmitted by one operator *TxSignalLink*. A maximum of 4000 *TxSignalLink* operators can be used in a design. The number of *RxSignalLink* operators is not restricted. Resource *SignalChannel* is visible in the *Resources* dialog.

**** EventID stands for maximal amount of events supported by the software for the particular platform. EventPort represents the event channel. One event channel can host up to 16 events.

***** Resource *ImageChannel* allows to connect *TxImageLink* operators with *RxImageLink* operators. Each operator *TxImageLink* needs one resource *ImageChannel* exclusively. Each resource *ImageChannel* can be connected to exactly one operator *RxImageLink*, i.e., a maximum of 1024 *TxImageLink* and 1024 *RxImageLink* operators can be used in one design. Resource *ImageChannel* is visible in the *Resources* dialog.

Table A.6. List of Device Resources microEnable 5 ironman

## A.2.3. LightBridge and microEnable 5 marathon

| Resource | mE5 marathon VCX-QP | mE5 marathon VF2 | mE5 marathon VCL | mE5 marathon VCLx | LightBridge VCL |
|---|---|---|---|---|---|
| Camera Port | 4 | 2 | 2 | 2 | 2 |
| CameraControl | - | - | 2 | 2 | 2 |
| DMA | 4 | 4 | 4 | 4 | 4 |
| GPO* | 10 OUT | 10 OUT | 10 OUT | 10 OUT | 6 OUT |
| GPI** | 12 IN | 12 IN | 12 IN | 12 IN | 8 IN |
| RAM | 4 x 512 MiB | 4 x 512 MiB | 4 x 512 MiB | 4 x 512 MiB | 4 x 512 MiB |
| LED Ports* | 4 | 4 | 2 | 2 | 2 |
| SignalChannel*** | 4000 | 4000 | 4000 | 4000 | 4000 |
| EventPort**** | 14 | 14 | 10 | 10 | 10 |
| EventID**** | 64 | 64 | 64 | 64 | 64 |
| ImageChannel***** | 1024 | 1024 | 1024 | 1024 | 1024 |

* These resources are not visible in the *Resources* dialog. They are controlled via operators.

| Resource | mE5 marathon VCX-QP | mE5 marathon VF2 | mE5 marathon VCL | mE5 marathon VCLx | LightBridge VCL |
|---|---|---|---|---|---|
| ** GPI is not visible in the *Resources* dialog. The same resource can be used multiple times. The table lists the amount of GPI ports.<br><br>*** Resource *SignalChannel* allows to connect *TxSignalLink* operators with *RxSignalLink* operators. Each operator *TxSignalLink* needs one resource *SignalChannel* exclusively. Multiple operators *RxSignalLink* can use the same resource *SignalChannel*, i.e., multiple operators *RxSignalLink* can receive the signals transmitted by one operator *TxSignalLink*. A maximum of 4000 *TxSignalLink* operators can be used in a design. The number of *RxSignalLink* operators is not restricted. Resource *SignalChannel* is visible in the *Resources* dialog.<br><br>**** "EventID" stands for maximal amount of events supported by the software for the particular platform. "EventPort" represents the event channel. One event channel can host up to 16 events.<br><br>***** Resource *ImageChannel* allows to connect *TxImageLink* operators with *RxImageLink* operators. Each operator *TxImageLink* needs one resource *ImageChannel* exclusively. Each resource *ImageChannel* can be connected to exactly one operator *RxImageLink*, i.e., a maximum of 1024 *TxImageLink* and 1024 *RxImageLink* operators can be used in one design. Resource *ImageChannel* is visible in the *Resources* dialog. | | | | | |

Table A.7. List of Device Resources LightBridge and microEnable 5 marathon

## A.2.4. imaFlex CXP-12 Quad and imaFlex CXP-12 Penta

| Resource | imaFlex CXP-12 Quad | imaFlex CXP-12 Penta |
|---|---|---|
| Camera Port | 4 | 5 |
| CxpStatusPort | 4 | 5 |
| CxpRxTriggerPort | 4 | 5 |
| CxpTxTriggerPort | 4 | 5 |
| DMA | 4 | 5 |
| DmaFromHostPort | 1 | 1 |
| GPO* | 10 OUT | 12 OUT |
| GPI** | 12 IN | 12 IN |
| LED Ports* | 6 | 6 |
| SignalChannel*** | 4000 | 4000 |
| EventPort**** | 32 | 32 |
| EventID**** | 64 | 64 |
| ImageChannel***** | 1024 | 1024 |
| RAM****** | from 1 x 1.5 GiB to 8 x 192 MiB | from 1 x 2.5 GiB to 8 x 320 MiB |
| * These resources are not visible in the *Resources* dialog. They are controlled via operators.<br><br>** GPI is not visible in the *Resources* dialog. The same resource can be used multiple times. The table lists the amount of GPI ports.<br><br>*** Resource *SignalChannel* allows to connect *TxSignalLink* operators with *RxSignalLink* operators. Each operator *TxSignalLink* needs one resource *SignalChannel* exclusively. Multiple operators *RxSignalLink* can use the same resource *SignalChannel*, i.e., multiple operators *RxSignalLink* can receive the signals transmitted by one operator *TxSignalLink*. A maximum of 4000 *TxSignalLink* operators can be used in a design. The number of *RxSignalLink* operators is not restricted. Resource *SignalChannel* is visible in the *Resources* dialog. | | |

| Resource | imaFlex CXP-12 Quad | imaFlex CXP-12 Penta |
|---|---|---|
| **** "EventID" stands for maximal amount of events supported by the software for the particular platform. "EventPort" represents the event channel. One event channel can host up to 16 events.<br><br>***** Resource *ImageChannel* allows to connect *TxImageLink* operators with *RxImageLink* operators. Each operator *TxImageLink* needs one resource *ImageChannel* exclusively. Each resource *ImageChannel* can be connected to exactly one operator *RxImageLink*, i.e., a maximum of 1024 *TxImageLink* and 1024 *RxImageLink* operators can be used in one design. Resource *ImageChannel* is visible in the *Resources* dialog.<br><br>****** RAM interface is shared across all RAM based operators in bandwidth and size. See section Shared Memory Concept for more details. | | |

Table A.8. List of Device Resources imaFlex CXP-12 Quad and imaFlex CXP-12 Penta

# A.3. Shared Memory Concept

## A.3.1. microEnable 5 marathon and LightBridge

The platforms microEnable 5 marathon and LightBridge are assembled with only one physical RAM bank (the size of which is platform-specific). This single physical bank is formatted into 4 non-overlapping memory regions. These 4 regions are represented inside VisualApplets as 4 virtual RAM banks. When an operator reserves a RAM resource, it is using a virtual RAM bank which maps to an exclusive non-overlapping memory region inside the physical RAM.

The RAM bandwidth, however, is shared between all RAM based operators in a design. When a design utilizes all 4 RAM resources, each of the 4 RAM based operators can have up to 1.6 GB/s exclusive bandwidth, minus the efficiency factor of that particular operator. When only one RAM based operator is used in the design, this operator gets the total bandwidth of 6.4 GB/s. When 2 operators are used, each of the two operators gets half the total bandwidth, etc.



**Bandwidth per Operator**

The on-board RAM provides 6.4GB/s total bandwidth. The bandwidth available for an individual RAM based operator is the total bandwidth divided by the number of all instantiated RAM based operators in the design.

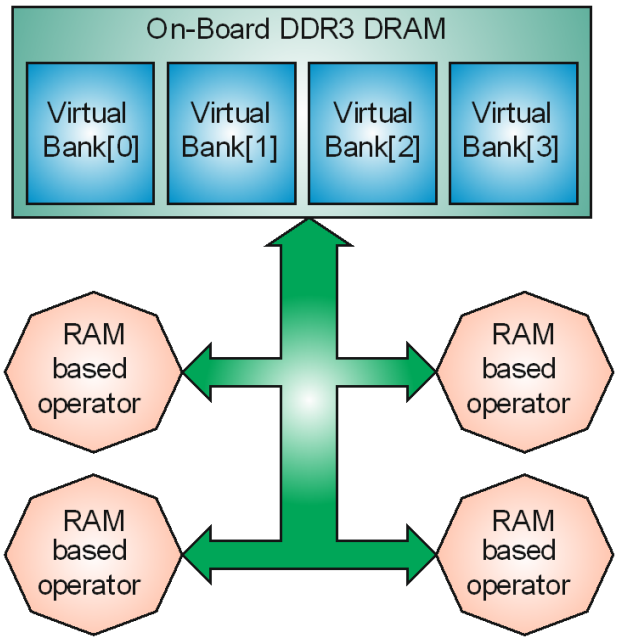



Figure A.1. RAM architecture

This RAM architecture needs to be considered when designing with RAM based operators.

Due to the shared bandwidth architecture, the applet developer should utilize all 256 bits of the operator's memory interface (RAM Data Width) to achieve maximal throughput through the memory interface when using multiple RAM based operators even though the single RAM operator needs less bandwidth on its input.

## A.3.2. imaFlex CXP-12 Quad and imaFlex CXP-12 Penta

The platforms imaFlex CXP-12 Quad and imaFlex CXP-12 Penta are assembled with only one physical RAM bank (the size of which is platform-specific). This single physical bank is dynamically formatted into non-overlapping regions depending on the amount of used RAM-based VisualApplets operators inside the applet. Those regions are represented inside VisualApplets as virtual RAM banks. When an operator reserves a RAM resource, it is using a virtual RAM bank which maps to an exclusive non-overlapping memory region inside the physical RAM. Up to 8 non-overlapping regions can be defined on the imaFlex platforms. When only 1 RAM operator is used, the operator gets the complete RAM size of the platform. The allocated size for each operator is reduced for each operator used in the design. If 8 operators are used, each operator will allocated 1/8 of the platform memory size.

The RAM bandwidth, however, is shared between all RAM-based operators in a design. When a design utilizes all 8 RAM resources, each of the 8 RAM-based operators can have up to 1/8 GB/s exclusive bandwidth, minus the efficiency factor of that particular operator. When only one RAM-based operator is used in the design, this operator gets the total bandwidth of the platform. When 2 operators are used, each of the two operators gets half the total bandwidth, etc.

### A.3.2.1. RAM Size Distribution Across RAM Ports

Not allocated and thus not used RAM ports get always 0% of the memory size.

- 1 port is utilized: the utilized port gets 100%.
- 2 ports are utilized: both utilized ports get 50%.
- 3 ports are utilized: the port with the lowest resource ID number will get 50%, the other 2 utilized ports get 25%.
- 4 ports are utilized: all 4 ports get 25%.

- 5 ports are utilized: 3 ports with the lowest resource ID number will get 25%. The last 2 ports will get 12.5%.
- 6 ports are utilized: 2 ports with the lowest resource ID number will get 25%. The other 4 ports will get 12.5%.
- 7 ports are utilized: the port with the lowest resource ID number will get 25%. The other 6 ports will get 12.5%.
- 8 ports are utilized: all ports get 12.5%.



Ports with the lower RAM index will get larger allocation in case of asymmetric size partitioning. For example 3 RAM ports are used: 0, 1, 2. The port 0 gets 50% RAM size allocation. The ports 1 and 2 get both 25% each.

The RAM indexes do not need to be contiguous, their absolute order decides the allocation. The RAM index is a virtual number and has no impact on FPGA resource usage, when having gaps in ordering. For example 3 RAM ports are used: 1, 5, 7. The ports 1 gets 50% RAM size allocation. The ports 5 and 7 get both 25% each.

There is no advantage or disadvantage when allocating RAM indexes of the operators. Use either the automatic VisualApplets allocation or tune it manually, when a special design operator needs more RAM size than the other operators.

### A.3.2.2. RAM Bandwidth Distribution Across RAM Ports

The shared memory controller applies the Round & Robin algorithm and distributes the bandwidth evenly across all allocated ports. The algorithm is using the credits arbitration scheme: When a port gets active, it can stay active for the credit's clock cycles as long as the port provides new RAM jobs. Once a port is activated but has no more jobs, the port gets deactivated and the activation token jumps to the next port in line, which has request jobs pending. This way, the bandwidth is never wasted on idling. The credits are programmed by the firmware exclusively during synthesis of the applet depending on the amount of used RAM operators and can't be changed in the Framegrabber SDK/ during runtime. The user has no access to the credit's programming. While an active port is owning the RAM interface, it will have access to 100% bandwidth of the memory controller. When all 8 ports are used and are active evenly, the resulting bandwidth over time is 1/8 for each port. In all other cases, the load on the ports defines the actual average bandwidth for each port.

# Glossary

**Area of Interest (AOI)**

See Region of Interest.

**Block RAM**

See FPGA Internal Block RAM.

**Board**

A Basler hardware. Usually, a board is represented by a frame grabber. Boards might comprise multiple devices.

**Board ID Number**

An identification number of a board in a PC system. The number is not fixed to a specific hardware but has to be unique in a PC system.

**Camera Index**

The index of a camera connected to a frame grabber. The first camera will have index zero. Mind the difference between the camera index and the frame grabber camera port.
See also Camera Port.

**Camera Port**

The frame grabber connectors for cameras are called camera ports. They are numbered {0, 1, 2, ...} or enumerated {A, B, C, ... }. Depending on the interface one camera could be connected to multiple camera ports. Also, multiple cameras could be connected to one camera port.

**Camera Tap**

See Tap.

**Device**

A board can consist of multiple devices. Devices are numbered. The first device usually has number one.

**Direct Memory Access (DMA)**

A DMA transfer allows hardware subsystems within the computer to access the system memory independently of the central processing unit (CPU).

DMA is used for data transfer such as image data between a board e.g. a frame grabber and a memory of the host system. Data transfers can be established in multiple directions i.e. from a frame grabber to the PC (download) and from the PC to a frame grabber (upload). Multiple DMA channels may exist for one board. Control and configuration data usually do not use DMA channels.

**Distributed RAM**

See FPGA Distributed RAM.

**DMA Channel**

See DMA Index.

**DMA Index**

The index of a DMA transfer channel.
See also Direct Memory Access.

**FPGA Distributed RAM**

Logic cells of the FPGA which are used as memory. Also called LUT RAM

**FPGA Internal Block RAM**

Memory blocks in the FPGA. Each FPGA includes a limited number of Block RAMs.

**Frame Grabber RAM**

A memory on the frame grabber but not in the FPGA. Usually DRAM.

**Framegrabber SDK**

See Runtime.
Basler provides a runtime environment for frame grabbers, which is called **Framegrabber SDK**. For further information, see the documentation of the Framegrabber SDK in the Basler Product Documentation (BPD): https://docs.baslerweb.com/frame-grabbers/framegrabber-sdk.html

**Hardware Applet (HAP)**

A hardware applet file is a build VisualApplets project. It can be loaded to licensed hardware devices e.g. frame grabbers. A HAP includes the FPGA bitstream as well as the software interface to access data and parameter. A HAP can only be used on one target runtime defined before the build was executed.

**Host PC**

The computer the frame grabber or device is attached to.

**Least Significant Bit (LSB)**

The bit position in a binary value having the lowest value i.e. the right-most bit.

**Library**

See Operator Library.

**LUT RAM**

See FPGA Distributed RAM.

**Mebibyte (MiB)**

One Mebibyte (MiB) are 1.048.576 Byte.

**Megabyte (MB)**

One Megabyte (MByte or MB) are 1,000,000 Byte.

**Module**

An instantiated operator in a VisualApplets diagram. See Operator.

**Most Significant Bit (MSB)**

The bit position in a binary value having the greatest value i.e. the left-most bit.

**Operator**

Represents an image-, data-, or signal-processing functionality. Is organized in operator libraries.

**Operator Library**

Operator Libraries include operators. Operators from an operator library can be used in a VisualApplets diagram.

**PC RAM**

Memory on the PC. Usually system memory or main memory on the mainboard. Accessed from the frame grabber via **Direct Memory Access** .

**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.

**Random Access Memory (RAM)**

A memory which can be directly accessed for reading and writing in any random order.

**Region of Interest (ROI)**

A part a frame. Mostly rectengular and within the original image boundaries. Defined by coordinates. The frame grabber cuts the region of interest from the camera image. A region of interest might reduce or increase the required bandwidth.

**Runtime**

In the context of VisualApplets, runtime is the environment in which the built hardware applet is used in an application.

Basler provides a runtime environment for frame grabbers, which is called **Framegrabber SDK**. For further information, see the documentation of the Framegrabber SDK in the Basler Product Documentation (BPD): https://docs.baslerweb.com/frame-grabbers/framegrabber-sdk.html

**Sensor Tap**

See Tap.

**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 simulatiously from different locations on the sensor. The reconstruction of the frame is called sensor readout correction.

The Camera Link interface is also using multiple taps for image tranfer to increase the bandwidth. These taps are independent from the sensor taps.

**Target Runtime**

Defines the target operating system environment on which the Framegrabber SDK is installed, and thus on which the built hardware applet is executed. Example: Windows 64-bit.

**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.

**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 triger IO. The first output has index 1. Check mapping of output pins to logic outputs in the harware documentation.

**UltaRAM (URAM)**

A type of dedicated memory blocks. UltraRam elements provide large FPGA memory capacity, additional to the BRAM resources. Using UltraRam may help when running out of BRAM resources.

# Bibliography

[Ada95] James E. Adams, Jr.: *Interactions Between Colorplane Interpolation and Other Image Processing Functions in Electronic Photography*. Proc. SPIE 2416,144 1995.

[Bas13] Jörg Kunze: *White Paper DeBayering with the sprint`s Raw Enhanced Mode*. https://www.baslerweb.com/en/downloads/document-downloads/debayering-sprint, April 4th 2017 June, 2013.

[Bay76] B. E. Bayer, Eastman Kodak Company: *Color imaging array*. United States Patent, Patent Number: 3,971,065 July 20, 1976.

[Bur06] Wilhelm Burger, Mark James Burge: *Digitale Bildverarbeitung*. Springer-Verlag Berlin Heidelberg , 2. Edition 2006.

[Dal05] N. Dalal B. Triggs: *Histogram of Oriented Gradients for Human Detection* . CVPR '05 Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01 Pages 886-893 , 2005.

[Deb97] P. E. Debevec, J. Malik: *Recovering High Dynamic Range Radiance Maps from Photographs*. Proceedings of SIGGRAPH 97, Computer Graphics Proceedings, Annual Conference Series, pp. 369-378 , August 1997.

[Eit07] M. Eitz : *High dynamic range imaging and tonemapping*. http://cybertron.cg.tu-berlin.de/eitz/hdr, 26th April 2016 , January 2007.

[Fat02] R. Fattal et al.: *Gradient Domain High Dynamic Range Compression*. ACM Transactions on Graphics 21, 3, pp. 249–256 , (July 2002.

[Lar94] C.A. Laroche et al., Eastman Kodak Company: *Apparatus and method for adaptively interpolating a full color image utilizing chrominance gradients*. Proceedings of Pacific Graphics 2007, S. 382–390. IEEE December 1994.

[Mer07] Tom Mertens et al.: *Exposure Fusion*. United States Patent, Patent Number: 5,373,322 December 2007 .

[Par09] J. Park. et al. : *Lens Distortion Correction Using ideal Images Coordinates*. IEEE Transactions on Consumer Electronics (Volume:55 , Issue: 3 ) August 2009.

[Ope16a] OpenCV : *Camera Calibration With OpenCV*. http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html 27th April 2016.

[Ope16b] OpenCV : *Geometric Image Transformations*. http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html, 27th April 2016 27th April 2016.

[Rei02] E. Reinhard. et al.: *Photographic Tone reproduction for digital Images*. ACM Transactions on Graphics 21, 3, pp. 267–276 July 2002.

# Index

## Symbols

0D Protocol, 37
1D Protocol, 37
2D Protocol, 37

## A

ABS, 625
Accumulator, 592
ActionCommand, 1180
Adaptive Threshold, 429
ADD, 627
AND, 911
AppendImage, 1494
AppendImageDyn, 1497
AppendLine, 1499
AppendLineDyn, 1501
AppletProperties, 1156
ARCCOS, 629
ARCCOT, 632
ARCSIN, 635
ARCTAN, 638
Area Scan, 37
Arithmetics, 623
Averaging 3x3, 480

## B

Bandwidth, 35, 47
Base, 674
BaseGrayCamera, 1186
BaseRgbCamera, 1189
Basic Acquisition, 385
Basic Principles, 14
Bayer De-Mosaicing, 431
  biColor, 455
  Bilinear_RG_GB, 445, 449
  High Quality Linear 5x5, 436
  Laplace, 437
  Laroche, 440
  Laroche Modified, 444
  Linear 3x3, 436
  Linear 3x3 with White Balance, 437
  Linear 5x5 with White Balance, 437
  Nearest Neighbor, 434
BAYER3x3Linear, 800
BAYER5x5Linear, 803
Binarization, 430
Binarization, Threshold, 365
Blob, 769
Blob1D, 430
Blob2D, 430
Blob2D ROI Select, 431
Blob_Analysis_1D, 776
Blob_Analysis_2D, 791
BoardStatus, 1164
Bottleneck, 47
BRANCH, 677
Build, 112
Build Configuration, 113
Build Settings, 113, 227

## C

CAM, 70
Camera Connection, 48
CameraControl, 70, 1183
CameraGrayArea, 1212
CameraGrayAreaBase, 1214
CameraGrayAreaFull, 1216
CameraGrayAreaMedium, 1218
CameraGrayLine, 1220
CameraGrayLineBase, 1222
CameraGrayLineFull, 1224
CameraGrayLineMedium, 1226
CameraRgbArea, 1228
CameraRgbAreaBase, 1230
CameraRgbAreaMedium, 1232
CameraRgbLine, 1234
CameraRgbLineBase, 1236
CameraRgbLineMedium, 1238
Cameras, Multiple, 211
CASE, 914
CastBitWidth, 678
CastColorSpace, 681
CastKernel, 682
CastParallel, 686
CastType, 688
CLHSDualCamera, 1240
CLHSPulseIn, 1244
CLHSPulseOut, 1248
CLHSSingleCamera, 1252
Clipboard, 32
ClipHigh, 641
ClipLow, 643
Close (Morphology), 480
CMP_AgeB, 916
CMP_AgtB, 918
CMP_AleB, 920
CMP_AltB, 922
CMP_Equal, 924
CMP_NotEqual, 926
Co-Processor, 462, 462
CoefficientBuffer, 955
ColMax, 594
ColMin, 596
Color, 799
Color Plane Separation, 455
ColorTransform, 807
ColSum, 598
Compression, 820
CONST, 690
ConvertPixelFormat, 692
Coordinate_X, 695
Coordinate_Y, 697
COS, 645
COT, 648
Count, 600
COUNTER, 1365
CreateBlankImage, 1507
CustomSignalOperator, 1367
CutImage, 1503
CutLine, 1505

CxpAcquisitionStatus, 1277
CxpCamera, 1257
CxpCameraMultiTap, 1265
CXPDualCamera, 1293
CxpPortStatus, 1278
CXPQuadCamera, 1301
CxpRxTrigger, 1288
CXPSingleCamera, 1309
CxpTxTrigger, 1290

## D

Data Flow, 35
Data Structure, 37
Dead Pixel Replacement, 556
Deadlock, 383
Deadlocks, 45
Debugging, 836
DelayToSignal, 1399
Depth From Focus
  Loop, 538
Design Rules Check, 74
Device Resources, 70
DigIOPort, 1317
DILATE, 884
Distortion Correction, 484, 495, 499
DIV, 651
DMA, 70
DMA Channels, Multiple, 365
DmaFromPC, 1318
DmaToPC, 1321
Downsampling, 483, 484
Downscale, 1402
DRC, 74
Dummy, 699
DynamicROI, 700

## E

Eccentricity, 501
Editing a Design, 28
EnumParamReference, 1063
EnumParamTranslator, 1068
EnumVariable, 1075
Equation to Implementation, 377
ERODE, 886
Event, 70
EventDataToHost, 709
EventSource, 70
EventToHost, 703
EventToSignal, 1405
Example-Image Statistics, 473
Example-Trigger Statistics, 473
Examples, 412, 416, 577, 583, 586
  Adaptive Threshold, 429
  Auto Threshold Mean, 429, 429
  Averaging 3x3, 480
  Bayer De-Mosaicing, 431
  Blob1D, 430
  Blob2D, 430
  Blob2D ROI Select, 431
  Close (Morphology), 480
  Co-Processor, 462, 462
  Color Plane Separation, 455
  Dead Pixel Replacement, 556
  Downsampling, 483, 484
  FFT, 416
  Filter Basics, 481
  Filter for Line Scan, 482
  Filter Sub Kernels, 482
  Flat Field Correction, 557, 557
  Gaussian Filter 5x5, 481
  Hardware Test, 463
  High Boost Sharpening Filter, 483
  Hir or Miss, 480
  HSL Color Classification, 461
  Image Dimension Test, 471
  Image Flow Control, 473
  Image Grayscale Scope, 472
  Image Injector, 472
  Image Monitoring, 472
  Image Timing Generator, 471
  JPEG, 418, 422, 425
  Kirsch Filter, 479
  Laplace Filter, 483
  Laser Pointer Detection, 428
  Laser Triangulation, 428
  Lookup Table, 535, 535, 536, 536
  Median Filter, 481
  Mirroring (Line), 504
  Morphological Edge, 478
  Motion Detection, 478
  Noise Reduction, 478
  Open (Morphology), 480
  Packbits Run Length Encoder, 428
  Parallel Filter, 482
  Roberts Cross Gradient, 479
  Run Length Encoder, 428
  Shading Correction (1D), 557, 558
  Shading Correction (2D), 557, 557
  Sobel Multi Gradient, 479
  Sobel X, 479
  Threshold, 430
  Trigger, 558, 559, 559, 559, 560, 560, 561, 561, 568, 570, 571, 573, 574, 575
  White Balancing, 462
ExpandLine, 1510
ExpandPixel, 1512
ExpandToKernel, 712
ExpandToParallel, 713
Exposure Fusion, 530

## F

Fast Fourier transform example, 416
FFT, 1577
FFT Example, 416
Filter, 882
Filter Basics, 481
FIRkernelNxM, 888
FIRoperatorNxM, 894
Flat Field Correction, 557, 557
FloatFieldParamReference, 1078
FloatParamReference, 1084
FloatParamSelector, 1148
FloatParamTranslator, 1089
FloatVariable, 1097

Flow Control, 38
FrameBufferMultiRoiDyn, 962
FrameBufferRandomRead, 486, 550, 969
FrameBufferRandomRead (imaFlex), 973
FrameEndToSignal, 1407
Framegrabber SDK, 120
FrameMax, 603
FrameMemory, 979
FrameMemoryRandomRd, 982
FrameMin, 605
FrameStartToSignal, 1409
FrameSum, 607
FullGrayCamera, 1203
FullRgbCamera, 1207

## G

Gaussian Filter 5x5, 481
Generate, 1411
Geometric Transformation, 486, 490, 493, 542
GetSignalStatus, 1417
GetStatus, 715
Getting Started, 5
Gnd, 1419
GPI, 1325
GPO, 1329

## H

HAP, 112, 225
HAP Merger, 214
Hardware, 213
Hardware Applet, 112
Hardware Platform, 1152
Hardware Test, 463
HDR
  Debevec, 520, 526
Help
  Context Sensitive, 24
  Operator Reference Access, 24
Hierarchical Box, 122
HierarchicalBox, 716
Histogram, 609
HitOrMiss, 897
HSI2RGB, 810
HSL Color Classification, 461
HWMULT, 1369

## I

IF, 928
Image Dimension Test, 471
Image Dimensions, 37
Image Flow Control, 473
Image Grayscale Scope, 472
Image Injector, 472
Image Moments, 490, 501
Image Monitoring, 472
Image Protocols, 37
Image Select, 365
Image Timing Generator, 471
ImageAnalyzer, 838
ImageBuffer, 985
ImageBufferMultiRoI, 990
ImageBufferMultiRoIDyn, 995
ImageBufferSC, 1000
ImageBufferSpatial, 1004
ImageBuffer_JPEG_Gray, 821
ImageFifo, 1008
ImageFlowControl, 872
ImageInjector, 862
ImageMonitor, 879
ImageNumber, 718
ImageSequence, 1012
ImageStatistics, 844
ImageTimingGenerator, 866
ImageValve, 1514
Infinite Source, 48
InsertImage, 1516
InsertLine, 1519
InsertPixel, 1522
IntFieldParamReference, 1100
IntFieldVariable, 1121
IntParamReference, 1105
IntParamSelector, 1144
IntParamTranslator, 1110
IntVariable, 1118
IsFirstPixel, 1524
IsLastPixel, 1526
IS_Equal, 931
IS_GreaterEqual, 933
IS_GreaterThan, 935
IS_InRange, 937
IS_LessEqual, 939
IS_LessThan, 941
IS_NotEqual, 943

## J

JPEG, 417, 422, 425
  multiple JPEG, 418
JPEG_Encoder, 830
JPEG_Encoder_Gray, 824

## K

KernelRemap, 720
Keystone Correction, 484, 495
Kirsch Filter, 479
KneeLUT, 1015

## L

Laplace Filter, 483
Laser Pointer Detection, 428
Laser Triangulation, 428
Latency Error, 49
LED, 1333
Library
  Accumulator, 592
  Arithmetics, 623
  Base, 674
  Blob, 769
  Color, 799
  Compression, 820
  Debugging, 836
  Filter, 882
  Hardware Platform, 1152
  Logic, 909
  Memory, 951

Parameters, 1049
Prototype, 1364
Signal, 1396
Synchronization, 1491
Transformation, 1576
LimitSignalWidth, 1422
Line Scan, 37
Line Shear, 505
LineBuffer (imaFlex), 1022
LineEndToSignal, 1425
LineMemory, 1027
LineMemoryRandomRd, 1031
LineNeighboursNx1, 899
LineStartToSignal, 1427
Link Properties, 64
LinkParamTranslator, 1128
LinkProperties, 1125
Logic, 909
Lookup Table, 535, 535, 536, 536
LUT, 1034

## M

M-type, 40
MAX, 901
MEDIAN, 902
Median Filter, 481
MediumGrayCamera, 1193
MediumRgbCamera, 1197
Memory, 951
MergeComponents, 722
MergeKernel, 725
MergeParallel, 727
MergePixel, 730
Metadata, 64
microDisplay, 116
MIN, 903
Mirroring (Line), 504
Module Properties, 53
ModuloCount, 611
Motion Detection, 477
MULT, 653
Multiple DMA Channels, 365
Multiplexing, 381

## N

NativeTrgPortIn, 1336
NativeTrgPortInExt, 1337
NativeTrgPortOut, 1338
New Project, 28
NOP, 732
Normalized Cross Correlation, 551
NOT, 945
NumberOfHits, 904

## O

O-type, 40
Open (Morphology), 480
Open Project, 29
Operator, 40, 412, 415
ABS, 625
Accumulator, 577
ActionCommand, 1180
ADD, 627
AND, 911
AppendImage, 1494
AppendImageDyn, 1497
AppendLine, 1499
AppendLineDyn, 1501
AppletProperties, 1156
ARCCOS, 629
ARCCOT, 632
ARCSIN, 635
ARCTAN, 638
BaseGrayCamera, 1186
BaseRgbCamera, 1189
BAYER3x3Linear, 800
BAYER5x5Linear, 803
Blob_Analysis_1D, 776
Blob_Analysis_2D, 791
BoardStatus, 1164
BRANCH, 677
CameraControl, 1183
CameraGrayArea, 1212
CameraGrayAreaBase, 1214
CameraGrayAreaFull, 1216
CameraGrayAreaMedium, 1218
CameraGrayLine, 1220
CameraGrayLineBase, 1222
CameraGrayLineFull, 1224
CameraGrayLineMedium, 1226
CameraRgbArea, 1228
CameraRgbAreaBase, 1230
CameraRgbAreaMedium, 1232
CameraRgbLine, 1234
CameraRgbLineBase, 1236
CameraRgbLineMedium, 1238
CASE, 914
CastBitWidth, 678
CastColorSpace, 681
CastKernel, 682
CastParallel, 686
CastType, 688
CLHSDualCamera, 1240
CLHSPulseIn, 1244
CLHSPulseOut, 1248
CLHSSingleCamera, 1252
ClipHigh, 641
ClipLow, 643
CMP_AgeB, 916
CMP_AgtB, 918
CMP_AleB, 920
CMP_AltB, 922
CMP_Equal, 924
CMP_NotEqual, 926
CoefficientBuffer, 955
ColMax, 594
ColMin, 596
ColorTransform, 807
ColSum, 598
CONST, 690
ConvertPixelFormat, 692
Coordinate_X, 695
Coordinate_Y, 697
COS, 645

COT, 648
Count, 600
COUNTER, 1365
CreateBlankImage, 1507
CustomSignalOperator, 1367
CutImage, 1503
CutLine, 1505
CxpAcquisitionStatus, 1277
CxpCamera, 1257
CxpCameraMultiTap, 1265
CXPDualCamera, 1293
CxpPortStatus, 1278
CXPQuadCamera, 1301
CxpRxTrigger, 1288
CXPSingleCamera, 1309
CxpTxTrigger, 1290
DelayToSignal, 1399
DigIOPort, 1317
DILATE, 884
DIV, 651
DmaFromPC, 1318
DmaToPC, 1321
Downscale, 1402
Dummy, 699
dynamic append and cut, 577
DynamicROI, 700
EnumParamReference, 1063
EnumParamTranslator, 1068
EnumVariable, 1075
ERODE, 886
EventDataToHost, 709
EventToHost, 703
EventToSignal, 1405
ExpandLine, 1510
ExpandPixel, 1512
ExpandToKernel, 712
ExpandToParallel, 713
FFT, 1577
FIRkernelNxM, 888
FIRoperatorNxM, 894
FloatFieldParamReference, 1078
FloatParamReference, 1084
FloatParamSelector, 1148
FloatParamTranslator, 1089
FloatVariable, 1097
FrameBufferMultiRoiDyn, 962
FrameBufferRandomRead, 969
FrameBufferRandomRead (imaFlex), 973
FrameEndToSignal, 1407
FrameMax, 603
FrameMemory, 979
FrameMemoryRandomRd, 982
FrameMin, 605
FrameStartToSignal, 1409
FrameSum, 607
FullGrayCamera, 1203
FullRgbCamera, 1207
Generate, 1411
GetSignalStatus, 1417
GetStatus, 715
Gnd, 1419
GPI, 1325
GPO, 1329
HierarchicalBox, 716
Histogram, 609
HitOrMiss, 897
HSI2RGB, 810
HWMULT, 1369
IF, 928
ImageAnalyzer, 838
ImageBuffer, 985
ImageBufferMultiRoI, 990
ImageBufferMultiRoIDyn, 995
ImageBufferSC, 1000
ImageBufferSpatial, 1004
ImageBuffer_JPEG_Gray, 821
ImageFifo, 1008
ImageFlowControl, 872
ImageInjector, 862
ImageMonitor, 879
ImageNumber, 718
ImageSequence, 1012
ImageStatistics, 844
ImageTimingGenerator, 866
ImageValve, 1514
InsertImage, 1516
InsertLine, 1519
InsertPixel, 1522
IntFieldParamReference, 1100
IntFieldVariable, 1121
IntParamReference, 1105
IntParamSelector, 1144
IntParamTranslator, 1110
IntVariable, 1118
IsFirstPixel, 1524
IsLastPixel, 1526
IS_Equal, 931
IS_GreaterEqual, 933
IS_GreaterThan, 935
IS_InRange, 937
IS_LessEqual, 939
IS_LessThan, 941
IS_NotEqual, 943
JPEG_Encoder, 830
JPEG_Encoder_Gray, 824
KernelRemap, 720
KneeLUT, 1015
LED, 1333
LimitSignalWidth, 1422
LineBuffer (imaFlex), 1022
LineEndToSignal, 1425
LineMemory, 1027
LineMemoryRandomRd, 1031
LineNeighboursNx1, 899
LineStartToSignal, 1427
LinkParamTranslator, 1128
LinkProperties, 1125
LUT, 1034
MAX, 901
MEDIAN, 902
MediumGrayCamera, 1193
MediumRgbCamera, 1197
Memory, 578, 578
MergeComponents, 722

MergeKernel, 725
MergeParallel, 727
MergePixel, 730
MIN, 903
ModuloCount, 611
MULT, 653
NativeTrgPortIn, 1336
NativeTrgPortInExt, 1337
NativeTrgPortOut, 1338
NOP, 732
NOT, 945
NumberOfHits, 904
OR, 947
Overflow, 1574
PackbitsRLE, 1371
PARALLELdn, 734
PARALLELup, 737
PeriodToSignal, 1429
PixelNeighbours1xM, 906
PixelReplicator, 1531
PixelToImage, 1533
PixelToSignal, 1432
Polarity, 1434
PseudoRandomNumberGen, 740
PulseCounter, 1437
RamLUT, 1037
RamLUT (imaFlex), 1042
Register, 615
RemoveImage, 1536
RemoveLine, 1538
RemovePixel, 1540
ResourceReference, 1140
ReSyncToLine, 1545
RGB2HSI, 812
RGB2XYZ, 1392
RGB2YUV, 814
RND, 655
ROM, 1047
RowMax, 617
RowMin, 619
RowSum, 621
RS485, 1292
RsFlipFlop, 1439
RxImageLink, 1547
RxLink, 1339
RxSignalLink, 1441
SampleDn, 745
SampleUp, 748
SCALE, 657
Scope, 858
Select, 1443
SelectBitField, 750
SelectComponent, 752
SelectFromParallel, 754
SelectROI, 756
SelectSubKernel, 758
SetDimension, 760
SetSignalStatus, 1446
ShaftEncoder, 1449
ShaftEncoderCompensate, 1453
ShiftLeft, 660
ShiftRight, 662
signal, 412, 413, 413, 414, 414, 414, 579, 581, 581
SignalDebounce, 1456
SignalDelay, 1459
SignalEdge, 1462
SignalGate, 1464
SignalToDelay, 1469
SignalToEvent, 1362
SignalToPeriod, 1471
SignalToPixel, 1473
SignalToWidth, 1475
SignalWidth, 1477
SIN, 665
SORT, 908
SourceSelector, 1550
SplitComponents, 762
SplitImage, 1552
SplitKernel, 765
SplitLine, 1556
SplitParallel, 766
SQRT, 668
StreamAnalyzer, 852
StreamControl, 876
StringParamReference, 1135
SUB, 669
SYNC, 1558
synchronization, 580
SyncSignal, 1481
TAN, 671
Trash, 768
TrgBoxLine, 1373
TrgPortArea, 1341
TrgPortLine, 1345
TriggerIn, 1356
TriggerOut, 1358
trigonometric functions, 580
TxImageLink, 1571
TxLink, 1360
TxSignalLink, 1483
Vcc, 1485
WhiteBalance, 816
WhiteBalanceBayer, 818
WidthToSignal, 1488
XNOR, 949
XOR, 950
XYZ2LAB, 1395
OR, 947
Orientation, 501
Overflow, 1574
Overlay Blend, 377

## P

P-type, 40
PackbitsRLE, 1371
PARALLELdn, 734
Parallelism, 35
PARALLELup, 737
parameter
redirection, 583, 583, 584
selection, 584
translation, 585
Parameters, 1049

Parametrization, 53
PeriodToSignal, 1429
Pixel Order, 36
Pixel Plant, 214
PixelNeighbours1xM, 906
PixelReplicator, 493, 542, 1531
PixelToImage, 1533
PixelToSignal, 1432
Platform, 213
Polarity, 1434
Position Correction and Defect Detection Using Blob, 547
Print, 244
Print Inspection, 547, 550
Process, 211
Process Intercommunication, 212
Project Description, 225
Prototype, 1364
PseudoRandomNumberGen, 740
PulseCounter, 1437

## R

RAM, 70
RamLUT, 1037
RamLUT (imaFlex), 1042
Register, 615
RemoveImage, 1536
RemoveLine, 1538
RemovePixel, 1540
ResourceReference, 1140
ReSyncToLine, 1545
Revision Control, 236, 238
RGB2HSI, 812
RGB2XYZ, 1392
RGB2YUV, 814
RND, 655
Rolling Average
  Loop Handling, 536
ROM, 1047
RowMax, 617
RowMin, 619
RowSum, 621
RS485, 1292
RsFlipFlop, 1439
Runtime, 113, 225
RxImageLink, 1547
RxLink, 70, 1339
RxSignalLink, 1441

## S

SampleDn, 745
SampleUp, 748
SCALE, 657
Scaling, 508
Scope, 858
Screenshot, 244
Script Collection (Tcl), 237
Search Module, 20
Select, 1443
SelectBitField, 750
SelectComponent, 752
SelectFromParallel, 754
SelectROI, 756
SelectSubKernel, 758
SetDimension, 760
SetSignalStatus, 1446
Shading Correction (1D), 557, 558
Shading Correction (2D), 557, 557
ShaftEncoder, 1449
ShaftEncoderCompensate, 1453
Shear, 505
ShiftLeft, 660
ShiftRight, 662
Signal, 1396
Signal Links, 51
Signal Protocol, 37
SignalDebounce, 1456
SignalDelay, 1459
SignalEdge, 1462
SignalGate, 1464
SignalToDelay, 1469
SignalToEvent, 1362
SignalToPeriod, 1471
SignalToPixel, 1473
SignalToWidth, 1475
SignalWidth, 1477
Simulation, 76
SIN, 665
Sobel Multi Gradient, 479
Sobel X, 479
SORT, 908
SourceSelector, 1550
SplitComponents, 762
SplitImage, 1552
SplitKernel, 765
SplitLine, 1556
SplitParallel, 766
SQRT, 668
Stitching, 383
StreamAnalyzer, 852
StreamControl, 876
StringParamReference, 1135
SUB, 669
Switch Cameras, 377
SYNC, 1558
Synchronization, 42, 1491
Synchronization (Tutorial), 377
SyncSignal, 1481
Synthesis, 112
Synthesis Settings, 227

## T

TAN, 671
Tap Geometry Sorting, 515
Target Hardware, 213
Target Runtime, 113, 225
Tcl, 236, 237, 238
Tcl Scripting, 236, 238
Template Matching, 551
Threshold, 365
Threshold Binarization, 430
Time-Out, 45
Transformation, 484, 495, 508, 1576
Trash, 768

TrgBoxLine, 1373
TrgPortArea, 1341
TrgPortLine, 1345
Trigger, 558, 559, 559, 559, 560, 560, 561, 561, 568, 570, 571, 573, 574, 575
  Process without DMA, 212
TriggerIn, 1356
TriggerOut, 70, 1358
TxImageLink, 1571
TxLink, 70, 1360
TxSignalLink, 70, 1483

## V

Vcc, 1485
Versioning, 64, 225

## W

White Balancing, 437, 462
WhiteBalance, 816
WhiteBalanceBayer, 818
WidthToSignal, 1488
Workflow, 15

## X

XILINX Installation Detected, 20
XNOR, 949
XOR, 950
XYZ2LAB, 1395