

TUTORIAL: S32DS ISP VISUAL GRAPH TOOL

S32 DESIGN STUDIO 3.1 or higher
with S32V2xx development package
and Vision extension package for S32V2xx



EXTERNAL USE



SECURE CONNECTIONS
FOR A SMARTER WORLD



WAIT!

Welcome to S32 Design Studio for Vision, Version 2.0

[GETTING STARTED](#)[DOCUMENTATION](#)[VIDEO](#)[SUPPORT](#)

QUICK LINKS

-S32DS- -VISION-

NEW Application Project
NEW Library Project
NEW Project from Example

-S32DS- -VGT-

NEW APEX2 Program Project
NEW APEX2 Kernel Project
NEW APEX2 Graph Project
NEW ISP Data Flow Project

-S32DS- -DDR-

NEW S32V DDR Configuration
Project

Continue training with video. Use training video resources from the Getting Started with the S32DS for Vision 2.0 collection case studies, if you need a more visually active experience. Browse real examples of successful device programming across the Platform of all Products using case studies and...

Video Guides

Quick and easy features video guides links on MP4 video case studies

Create a New APEX2 Project from Example

Create a New APEX2 Project
Create an APEX2 Project from Example
Debug an APEX2 Project using Emulator
Debug an APEX2 Project using Lauterbach TRACE32

Debug an A53 Project using GDB PEMicro Interface
Debug an A53 Project using GDB Remote Linux

INTRODUCTION

video resources from the Getting Started with the S32DS for Vision 2.0 collection case studies, if you need a more visually active experience. Browse real examples of successful device programming across the Platform of all Products using

Looking for Interactive Tutorial?

- You can view this tutorial as a video under the **VIDEO** tab of Getting Started page of S32 Design Studio

Prerequisite to the tutorial

- Knowing the S32V234 product
- Have an understanding of the ISP architecture
- Be familiar with the vision SDK software

Agenda

- Tutorial Overview
- Make an ISP graph (project)
 - Make a basic graph
 - Configure graph/block properties
 - Generate source code from graph
 - Validate graph
 - Generate source code
- Application code for ISP
 - Make an application
 - Compile
 - Run

Tutorial Overview:

1. To start with, we will make an ISP graph using *ISP Dataflow Project* option
 - Just to make it simple we will use the ISP kernels available in Vision SDK
 - Vision SDK provides many built-in Kernels readily available for user development
 - Once graph is made we will autogenerate code for ISP engine
2. Moving forward, we will use the code derived above and make a Linux application project
 - This application will integrate ISP code with the host to run together smoothly.

Complete application will take an image from camera, processes it in ISP and put the processed image in DDR buffers. Once the image is in DDR buffer, host(A53 core running Linux) will direct the display control unit(DCU) to display it on screen

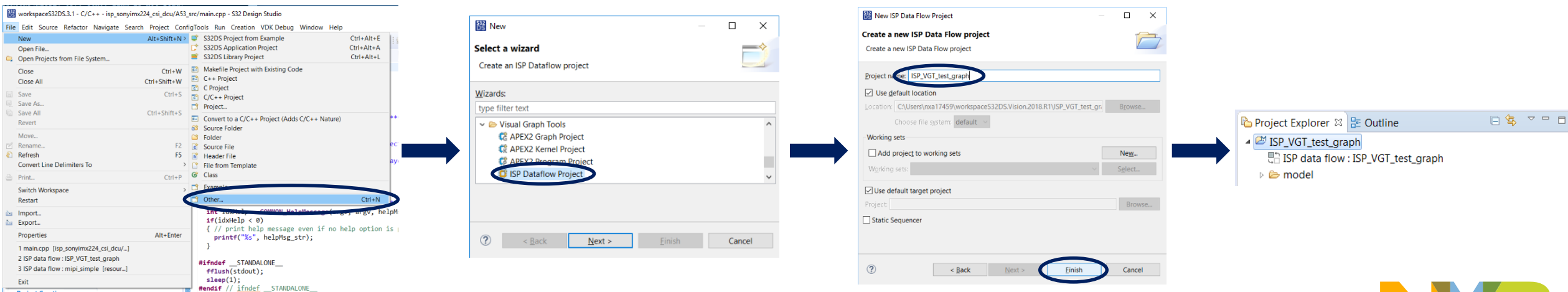
MAKE A GRAPH

First of all we will simply connect different blocks to make a basic graph



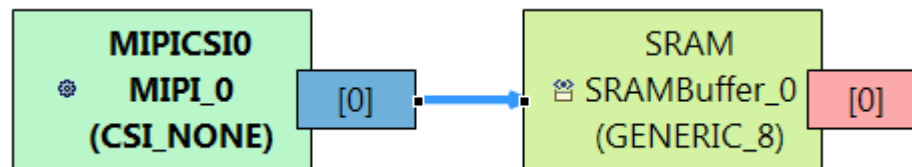
Make an ISP Dataflow Project

- We will make a simple graph that does following:
 - Grab an image from MIPI-CSI port >> Run Debayer kernel >> Transfer data to DDR buffer using Fast DMA
- Make a new *ISP Dataflow Project* named : **ISP_VGT_test_graph**

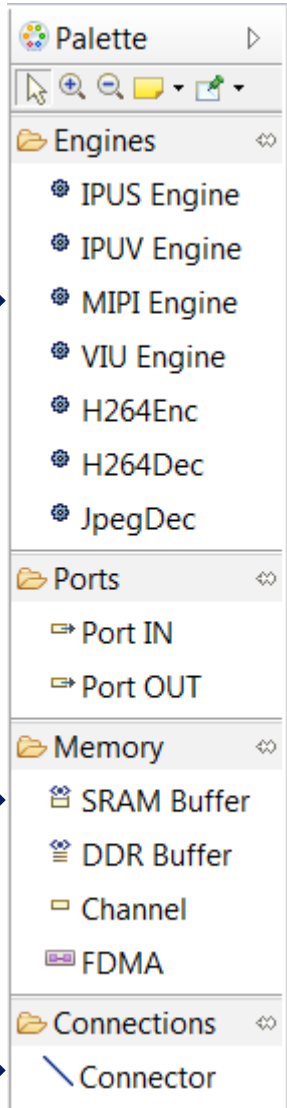


Make a graph

- From the **Palette** windows (right side of the S32DS window):
 - Camera is connected to MIPI-CSI interface. Hence, we will start our pipeline from MIPI interface
 - Select **MIPI Engine** block
 - Image lines fetched from MIPI-CSI will now go to SRAM
 - Select **SRAM Buffer**
 - Connect both using **Connector**

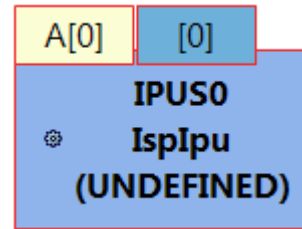


1 of 6

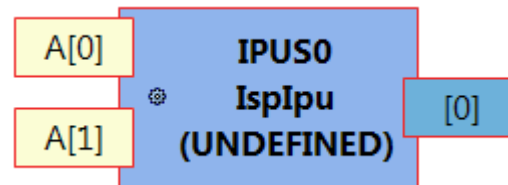


Make a graph

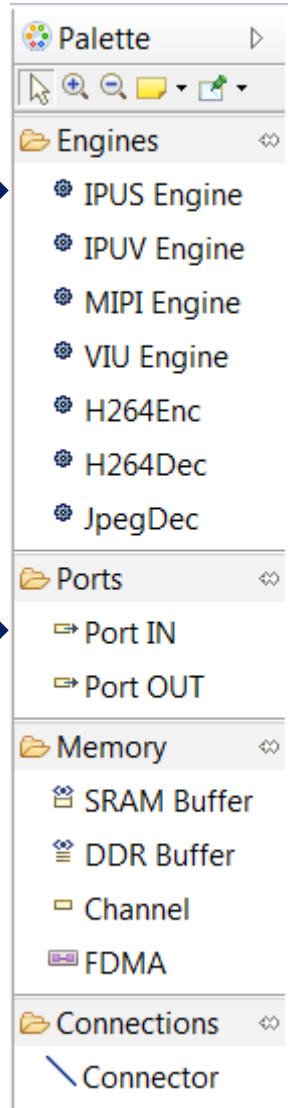
- Once necessary image lines are in SRAM, IPU(Image Processing Unit) engine grabs the image data and start processing it
 - So we will add **IPUS Engine**



- We will be using “debayering kernel: that requires 2 images lines from the SRAM buffer to start processing, so we will add one more input port
- Add a new input port to it: **Port In**



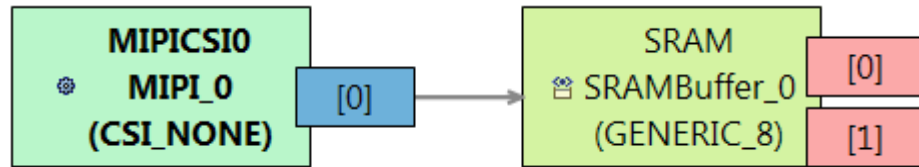
2 of 6



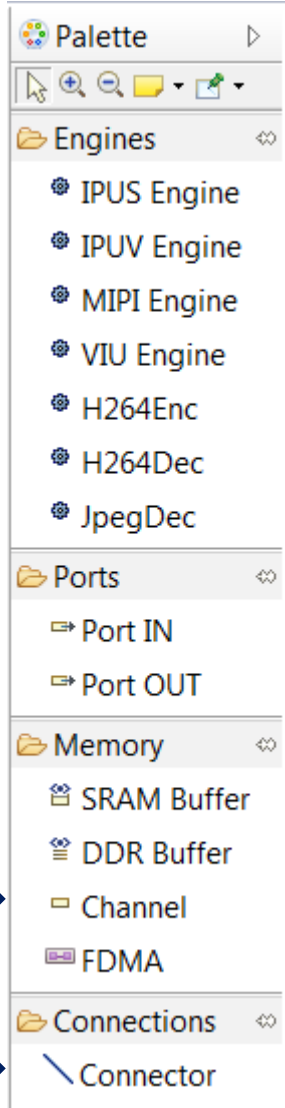
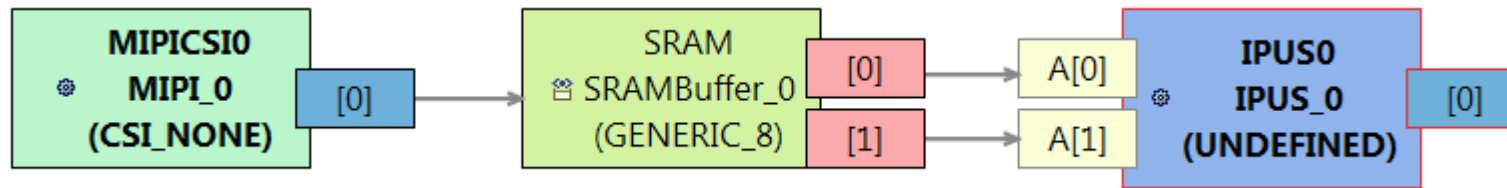
Make a graph

3 of 6

- For the same reason, add an output **Channel** to **SRAM Buffer**



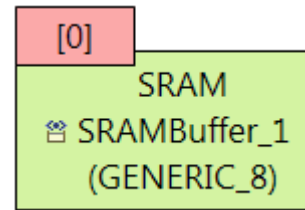
- **Connect** each **SRAM channel** with each **input port** of IPU engine



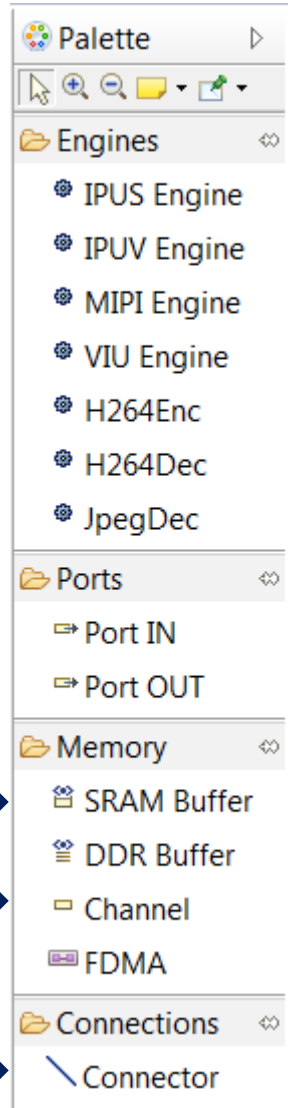
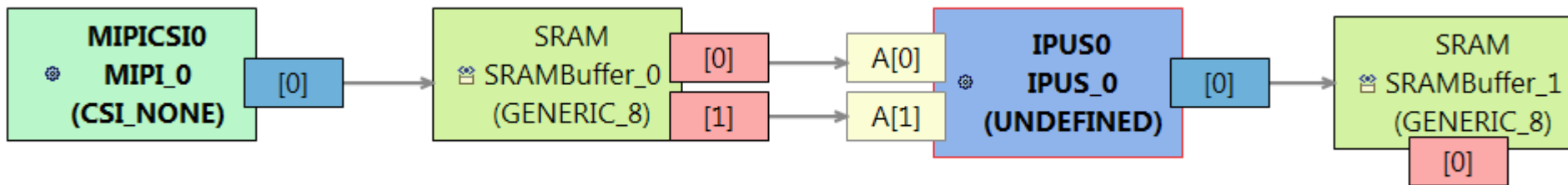
Make a graph

4 of 6

- Now, we will move processed data to SRAM
 - Add a new **SRAM Buffer**

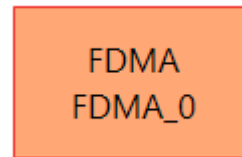


- **Connect IPU engine's output port with SRAM Buffer**

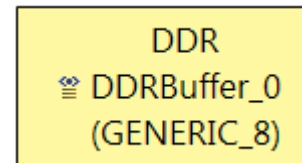


Make a graph

- We will move processed line to DDR buffer via Fast DMA
 - Add a new Fast DMA block: **FDMA**

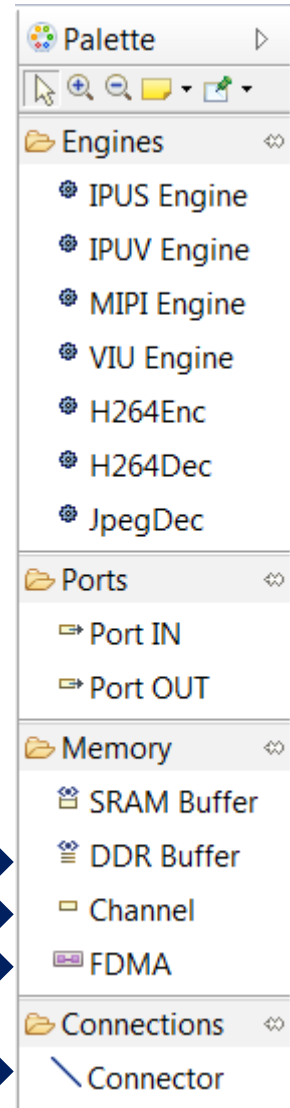


- Add a new **DDR Buffer** block



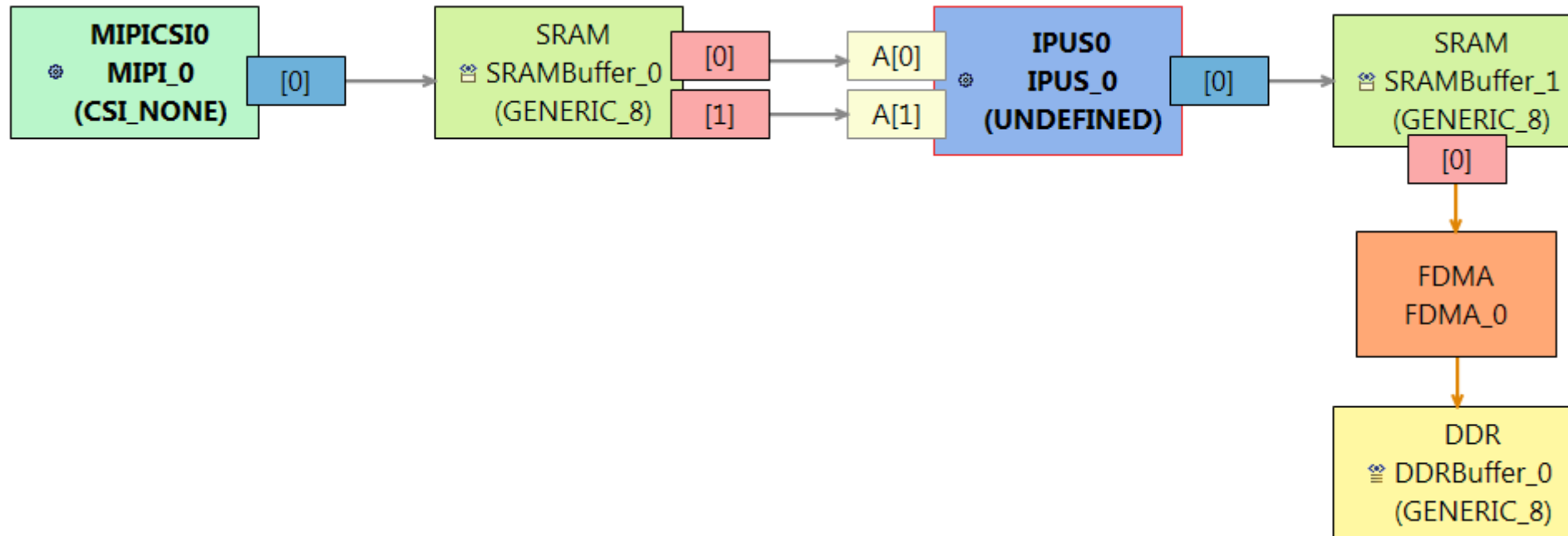
- Connect **Output Channel** of **SRMA Buffer** to **FDMA** and **FDMA** to **DDR Buffer**

5 of 6



Make a graph

- This is how your graph should now look like:



The basic ISP graph is READY!

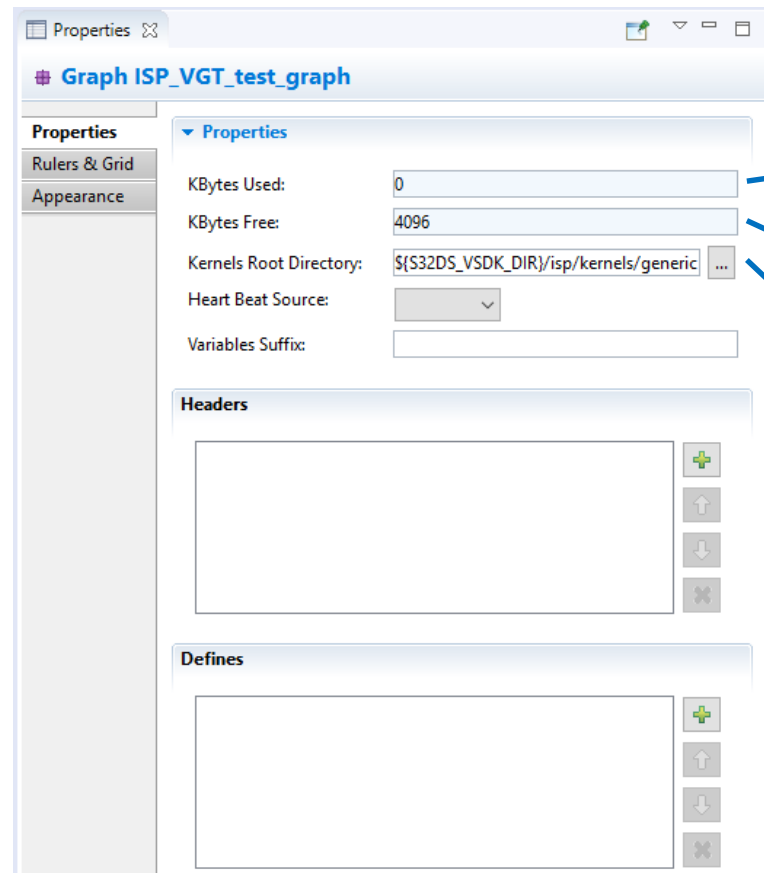
CONFIGURE THE BLOCKS PROPERTIES

The basic graph is not able to process image unless it is directed. So, we will configure graph blocks to process image information in particular manner



Graph properties

- **Click** on the white area of the graph and look at the **Properties**
- **Look at** (open if not visible) the properties window



Represent the space used in SRAM by the graph. In most cases, it should not go over 1024kB

Represents the space in SRAM unused by the graph. Although it covers all 4MBs of SRAM, remember that only **1MB** is optimized for the ISP usage.

Folder containing the ISP kernels (assembly files)

Configure the Blocks Properties

1 of 13

- **Select the MIPI Block**
- Again, **look at the properties window**

The screenshot displays a software development environment with a data flow diagram and a properties window. The data flow diagram, titled "ISP data flow : ISP_VGT_test_graph", shows a sequence of blocks: a green "MIPICSIO MIPI_0 (CSI_NONE)" block, a green "SRAM SRAMBuffer_0 (GENERIC_8)" block with two red ports labeled "[0]" and "[1]", two yellow blocks labeled "A[0]" and "A[1]", a blue "IPUS0 IPUS_0 (UNDEFINED)" block, another green "SRAM SRAMBuffer_1 (GENERIC_8)" block with a red port labeled "[0]", an orange "FDMA FDMA_0" block, and a yellow "DDR DDRBuffer_0 (GENERIC_8)" block. A blue circle highlights the "MIPICSIO MIPI_0" block, and a blue arrow points from this circle to the properties window. The properties window, titled "MIPI Engine MIPI_0", shows the following settings:

- Name: MIPI_0
- Camera Type: CSI_NONE
- Engine Type: MIPICSIO
- Attached Cameras: 1
- Input Image Lines: 0
- Frame Done Channel Index: 0

The right side of the interface shows a "Palette" window with categories: Engines (IPUS Engine, IPUV Engine, MIPI Engine, VIU Engine, H264Enc, H264Dec, JpegDec), Ports (Port IN, Port OUT), Memory (SRAM Buffer, DDR Buffer, Channel, FDMA), and Connections (Connector).

Configure the Blocks Properties

- **MIPI Block** : **Configure** the properties like follow:

Choose the name that you want for the block (without space)

Select the type of camera you are using

Number of attached cameras with this MIPI port

MIPI-CSI port, where you have connected your camera. In this case, camera is connected to MIPI-CSI_0

Number of lines to be fetched my MIPI. Sony camera sensor has 729 lines

Properties

MIPI Engine MIPI_0

Base

Name: MIPI_0

Properties

Camera Type: CSI_SONY224

Engine Type: MIPICSI0

Attached Cameras: 1

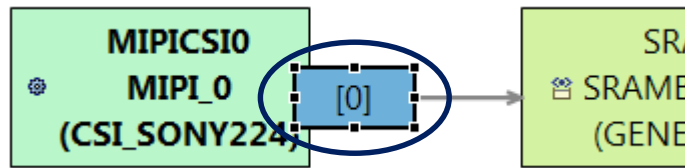
Input Image Lines: 729

Frame Done Channel Index: 0

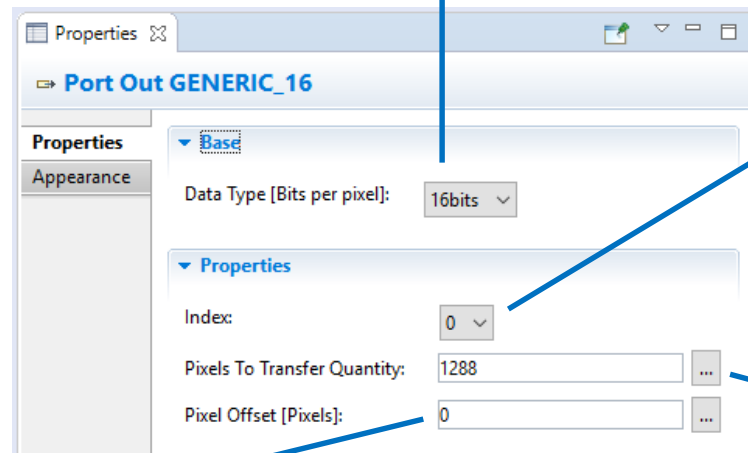


Configure the Blocks Properties

- **Select the Port OUT** of MIPI Engine



Choose datatype for pixel data. In this case, pixels coming from Sony camera are 12 bit, so we will choose 16 bit datatype



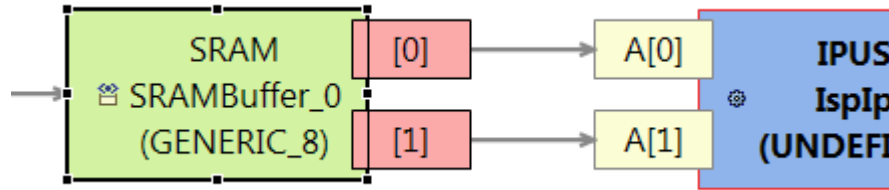
Corresponds to the virtual channel of MIPI-CSI

Number of pixel per line of an image
For Sony camera, each image line is 1288 pixel long

Number of Bytes to be skipped in each line. In this case, no bytes to be skipped



- **Select the SRAM Buffer**



The Stride has to be equal or superior to XSize. It is the number of bytes per line. It can also be incremented to extend the lines with some black pixel (0x0)

Number of Pixel per line

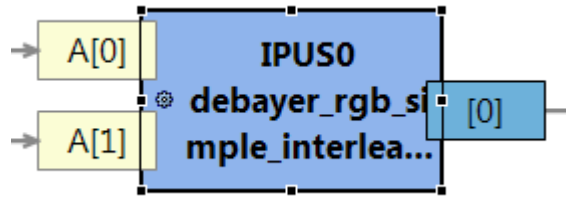
Since MIPI output port is 16 bit, we must choose 16 bit datatype again here

The screenshot shows the configuration window for 'SRAMBuffer_0'. The 'Name' field is 'SRAMBuffer_0'. Under the 'Properties' section, the following values are set: 'Data Type' is '16bits', 'Producer' is 'MIPI_0 out [0]', 'Line Increment' is '1', 'Stride [Bytes]' is '2576', 'XSize [Pixels]' is '1288', 'YSize [Lines]' is '32', and 'Fill Level' is '0'. Blue arrows point from the text boxes to these specific fields.

The size of the buffer in number of lines. The size of the buffer doesn't need to be very big in particular in this case where only one Fast DMA channel will be running



- **Select the IPU Engine**

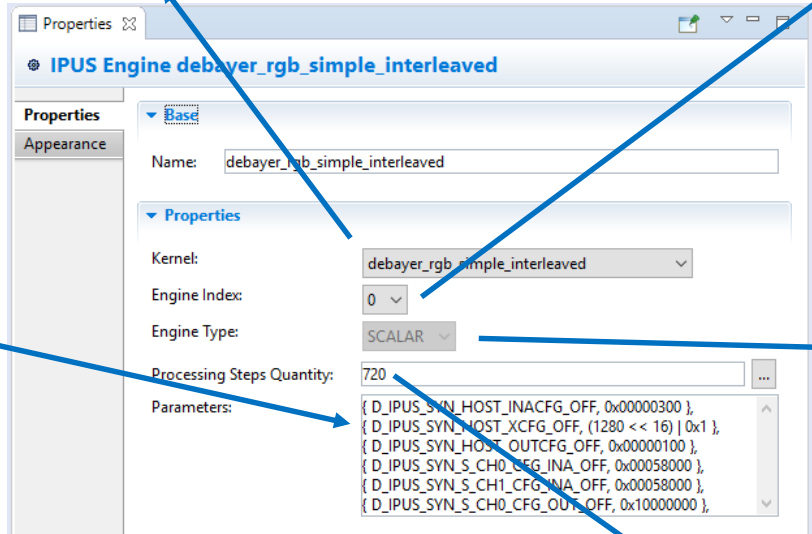


Select kernels from the menu. This folder is defined in the graph properties.

The number of the engine depends on what the kernel requires. In this case the kernel can run on any IPUS. We are selecting IPUS_0.

In this field, copy the following lines:

```
{ D_IPUS_SYN_HOST_INACFG_OFF, 0x00000300 },  
{ D_IPUS_SYN_HOST_XCFG_OFF, (1280 << 16) | 0x1 },  
{ D_IPUS_SYN_HOST_OUTCFG_OFF, 0x00000100 },  
{ D_IPUS_SYN_S_CH0_CFG_INA_OFF, 0x00058000 },  
{ D_IPUS_SYN_S_CH1_CFG_INA_OFF, 0x00058000 },  
{ D_IPUS_SYN_S_CH0_CFG_OUT_OFF, 0x10000000 },
```



The type of IPU (Scalar or Vector) depends for which engine the kernel has been developed for

Number of lines to be processed by the IPU.



Configure the Blocks Properties

- The description of the registers can be found in the RM
 - The example configurations for different kernels could be found in another graph(*s32ds_installation_directory\S32DS\s32v234_sdk\isp\graphs*)
- Here is its meaning:

```
{ D_IPUS_SYN_HOST_INACFG_OFF, 0x00000300 },
```

=> Enable InA[0] and InA[1] inputs

```
{ D_IPUS_SYN_HOST_XCFG_OFF, (1280 << 16) | 0x1 },
```

=> 1280 pixels per lines, pixel processed one at a time (XPOS incremented by 1 with “pixel done” kernel instruction)

```
{ D_IPUS_SYN_HOST_OUTCFG_OFF, 0x00000100 },
```

=> Enable OUT[0] output

```
{ D_IPUS_SYN_S_CH0_CFG_INA_OFF, 0x00058000 },
```

=> InA[0] configuration: 16 bits, streamed pixel not repeated, every pixels of a lines is used, no added padding on the image border

```
{ D_IPUS_SYN_S_CH1_CFG_INA_OFF, 0x00058000 },
```

=> InA[1] configuration: 16 bits, streamed pixel not repeated, every pixels of a lines is used, no added padding on the image border

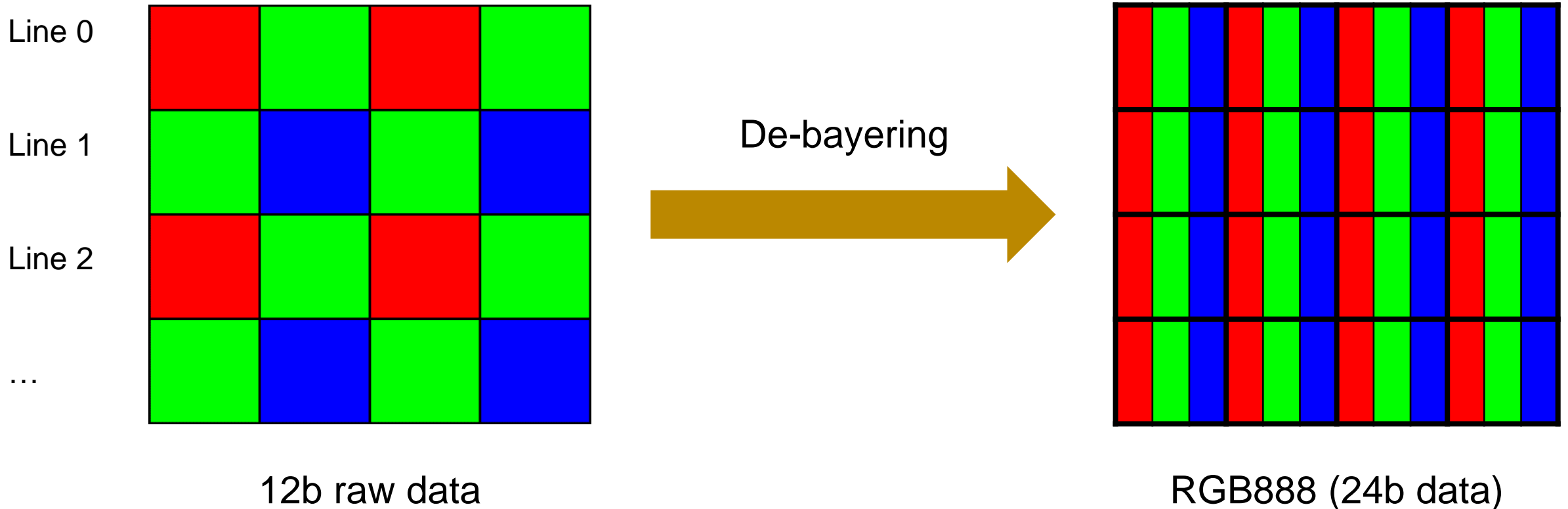
```
{ D_IPUS_SYN_S_CH0_CFG_OUT_OFF, 0x10000000 },
```

=> OUT[0] configuration: 8bits (the frame will be in RGB888: R, G and B will be outputted one per one)



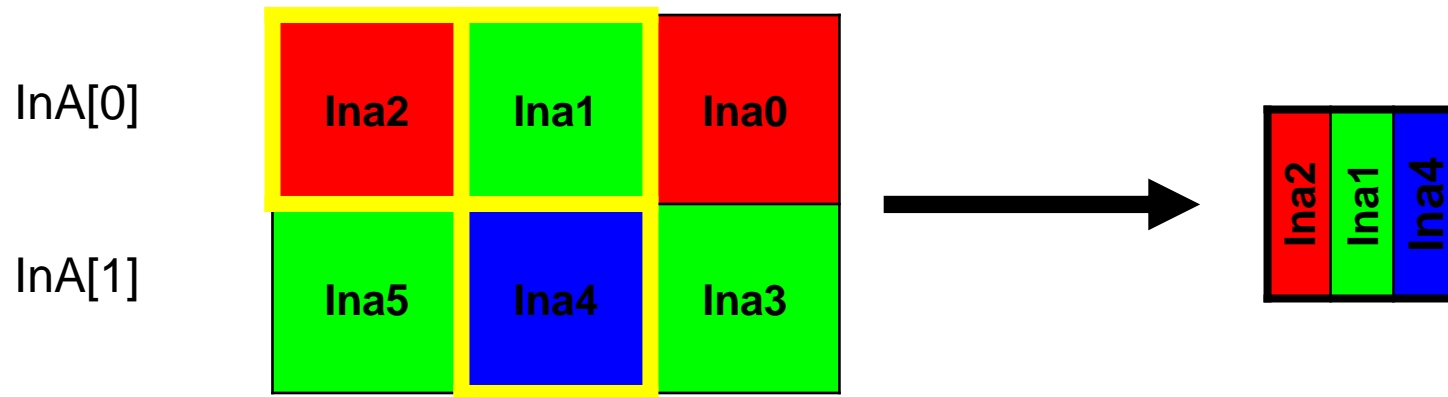
Explanations of debayer_rgb_simple_interleaved kernel

- Debayering is used to get the RGB value



Explanations of debayer_rgb_simple_interleaved kernel

- Simple debayering scheme used here is copying neighbouring pixel value to find RGB value of the one pixel



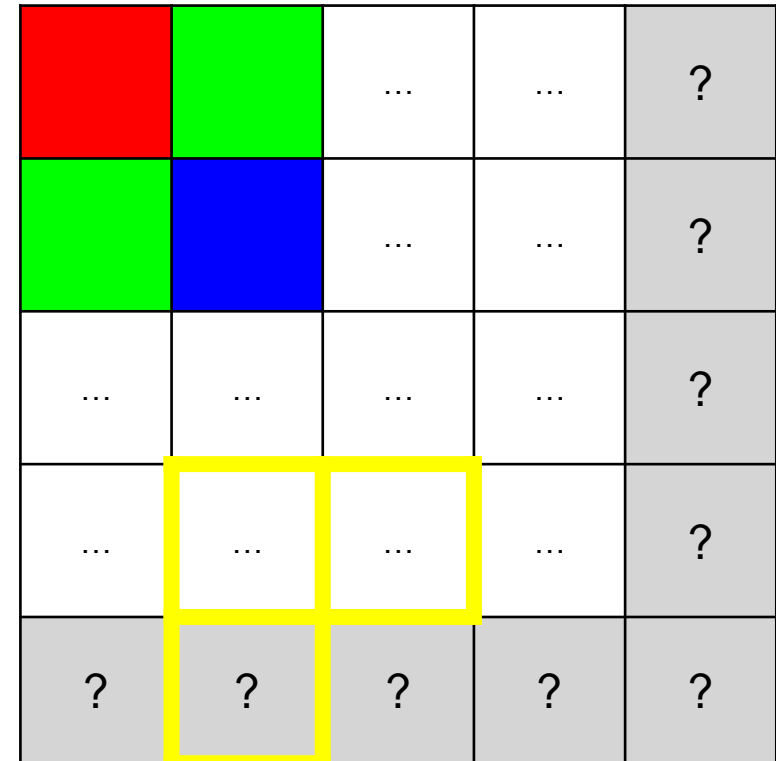
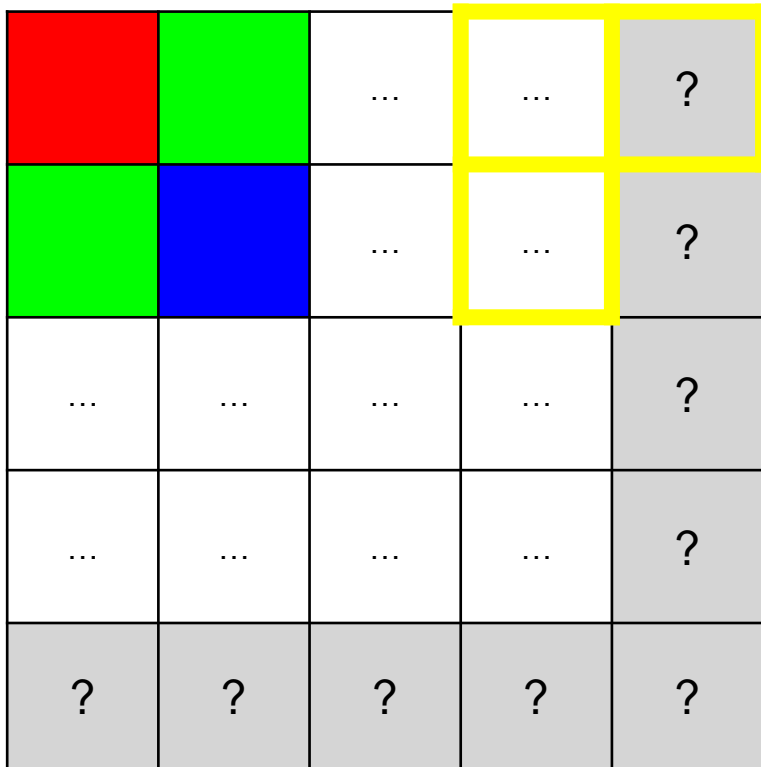
- With this scheme certain questions arise...
 - What happens when computing the last pixel of a line?
 - What happens when computing the last line of the frame?

Explanations of debayer_rgb_simple_interleaved kernel

- How to do on the border of the frame:



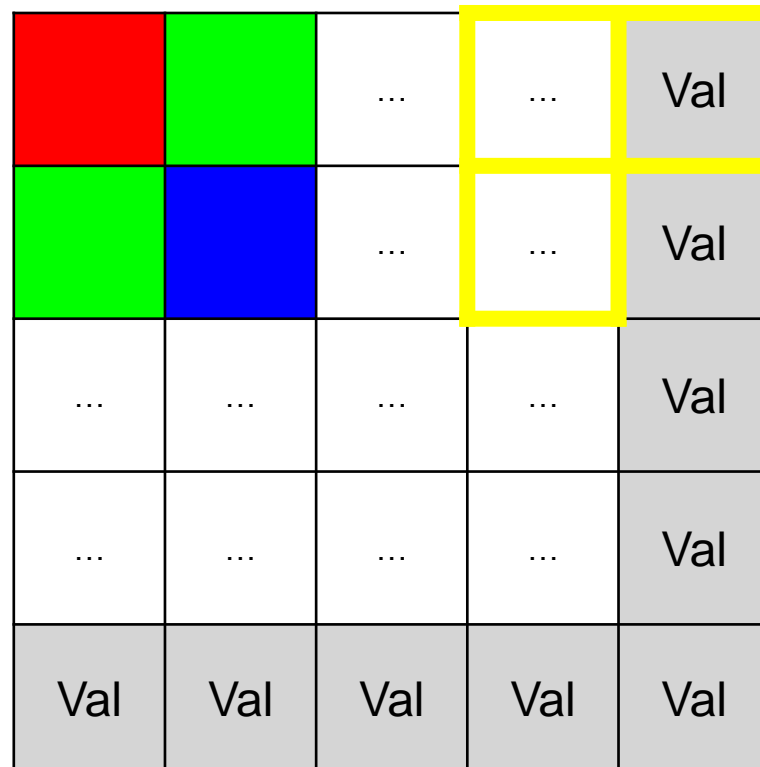
You cannot calculate the last pixel of the row and the last line



Explanations of debayer_rgb_simple_interleaved kernel

- Two solutions:

Solution 1: Use the Stream DMA to add lines (configuration in D_IPUS_SYN_S_CHx_CFG_INA_OFF)

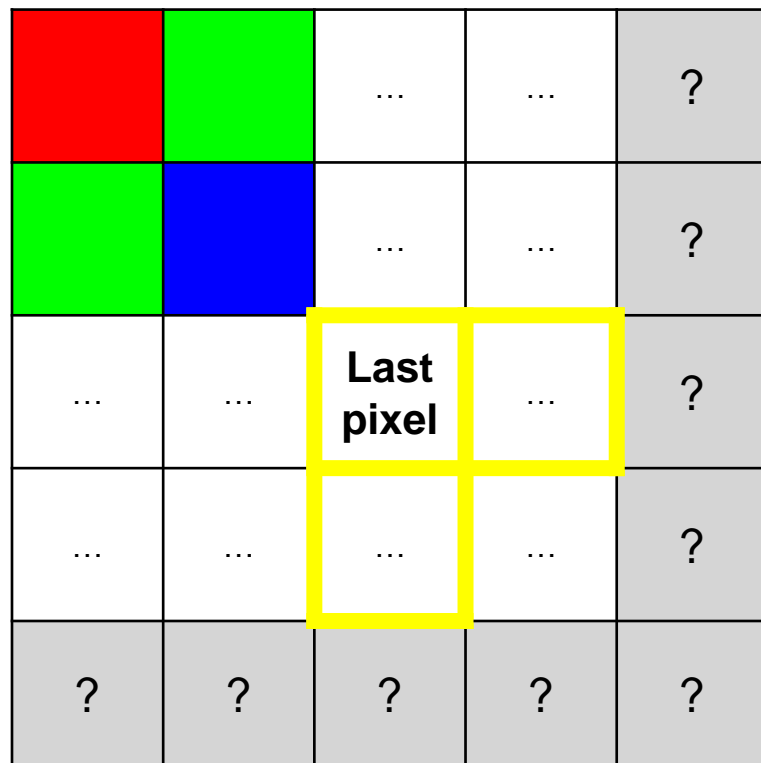


You can calculate the last pixel of the row and the last line

Explanations of debayer_rgb_simple_interleaved kernel

- Two solutions:

Solution 2: Not compute the last line and last row, decrease the resolution



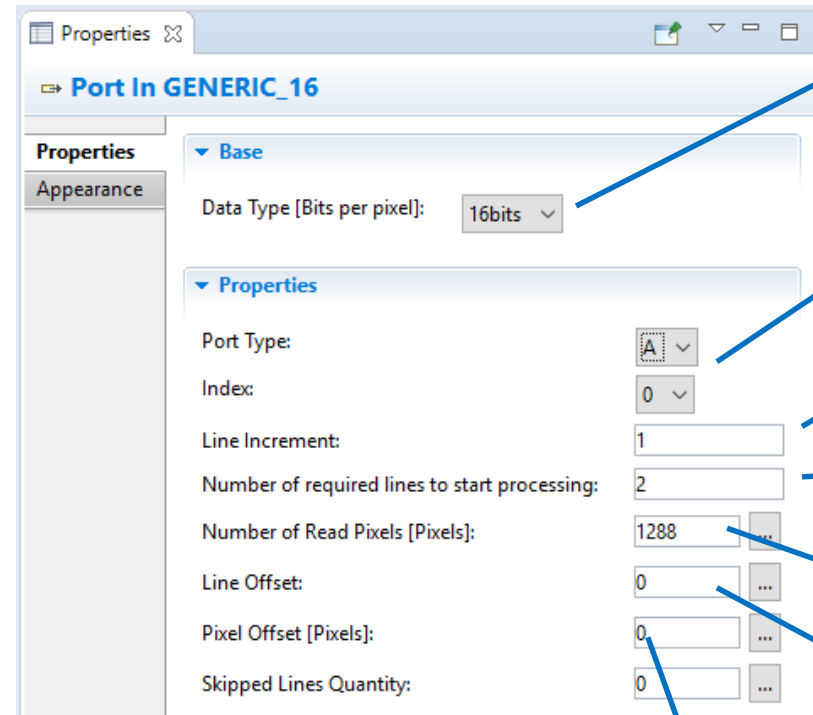
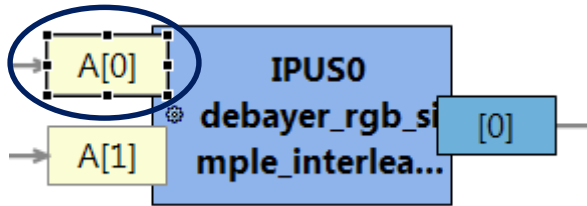
→ The last line and last row are sacrificed

In this example the solution 2 is chosen. The Sony camera has some extra lines and columns:
1296x726

Configure the Blocks Properties

8 of 13

- **Select the Port IN 0** of IPUS Engine



The pixels coming from the Sony camera are 12b

Port InA[0]

Go to next line when a line has finished to be processed(no jump)

The kernel requires 2 lines to work

1288 pixel per lines in input for the Sony camera

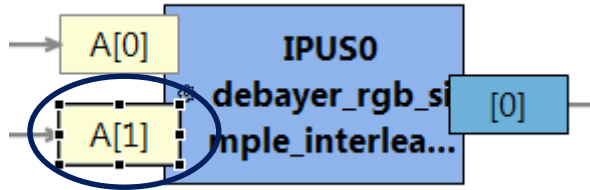
Start at the first pixel of the line

Start line 0



Configure the Blocks Properties

- **Select the Port IN 1 of IPUS Engine**



Port In GENERIC_16

Properties

Appearance

Base

Data Type [Bits per pixel]: 16bits

Properties

Port Type: A

Index: 1

Line Increment: 1

Number of required lines to start processing: 2

Number of Read Pixels [Pixels]: 1288

Line Offset: 1

Pixel Offset [Pixels]: 0

Skipped Lines Quantity: 0

Port InA[1]

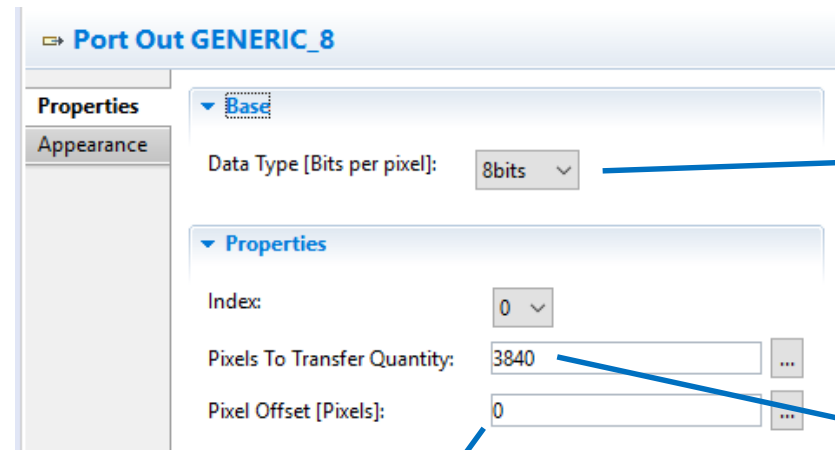
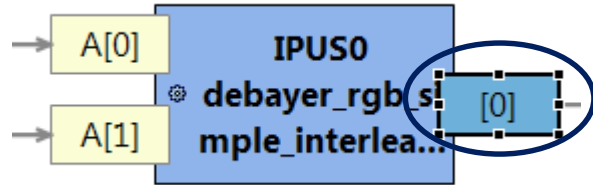
InA[1] is used to get the line below: start with **line 1**



Configure the Blocks Properties

10 of 13

- **Select the Port OUT** of IPUS Engine



The kernel outputs R, G and B successively to create RGB888 pixels

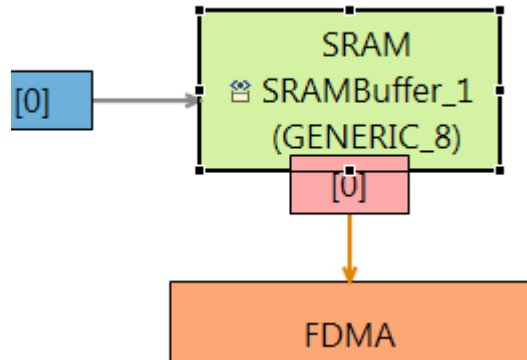
In this configuration we are cropping the image by ignoring the right column. We could add an offset to re-center the cropping

We want the output to be 1280 pixel wide (3x1280=3840)



Configure the Blocks Properties

- **Select the other SRAM Buffer**



The Stride has to be equal or superior to XSize. It is the number of bytes per line. It can also be incremented to extend the lines with some black pixel (0x0)

Number of Pixel per line

The screenshot shows the configuration window for 'SRAM Buffer SRAMBuffer_1'. The 'Properties' section is expanded, showing the following settings: Data Type: 8bits, Producer: debayer_rgb_simple_interleaved out [0], Line Increment: 1, Stride [Bytes]: 3840, XSize [Pixels]: 3840, YSize [Lines]: 16, and Fill Level: 0. Blue arrows point from the text boxes to the Stride [Bytes], XSize [Pixels], and YSize [Lines] fields.

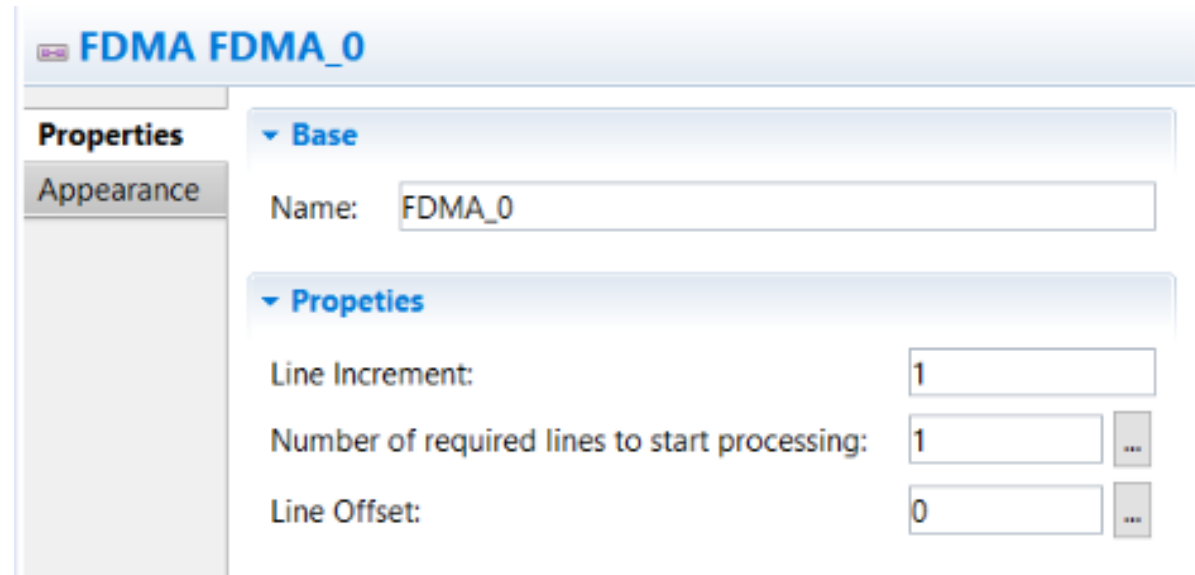
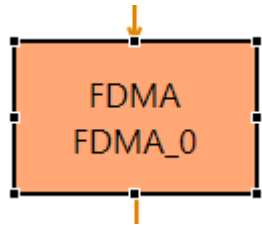
The size of the buffer in number of lines. The size of the buffer doesn't need to be very big in particular in this case where only one Fast DMA channel will be running



Configure the Blocks Properties

12 of 13

- **Select the FDMA** (Fast DMA block)

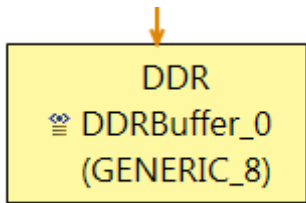


The screenshot shows the configuration window for the "FDMA FDMA_0" block. The window has a title bar with the text "FDMA FDMA_0". On the left side, there is a sidebar with two tabs: "Properties" (selected) and "Appearance". The main area of the window is divided into two sections: "Base" and "Properties".

- Base**
 - Name:
- Properties**
 - Line Increment:
 - Number of required lines to start processing: ...
 - Line Offset: ...

Configure the Blocks Properties

- **Select the DDR Buffer**



DDR Buffer DDRBuffer_0

Properties

Appearance

Base

Name: DDRBuffer_0

Properties

Data Type: 8bits

Producer: FDMA_0

Line Increment: 1

Stride [Bytes]: 3840

XSize [Pixels]: 3840

YSize [Lines]: 720

Stride = 3840 =
1280*3*1byte for RGB
image

Number of lines of the final frame

ISP graph is now
completely
READY!

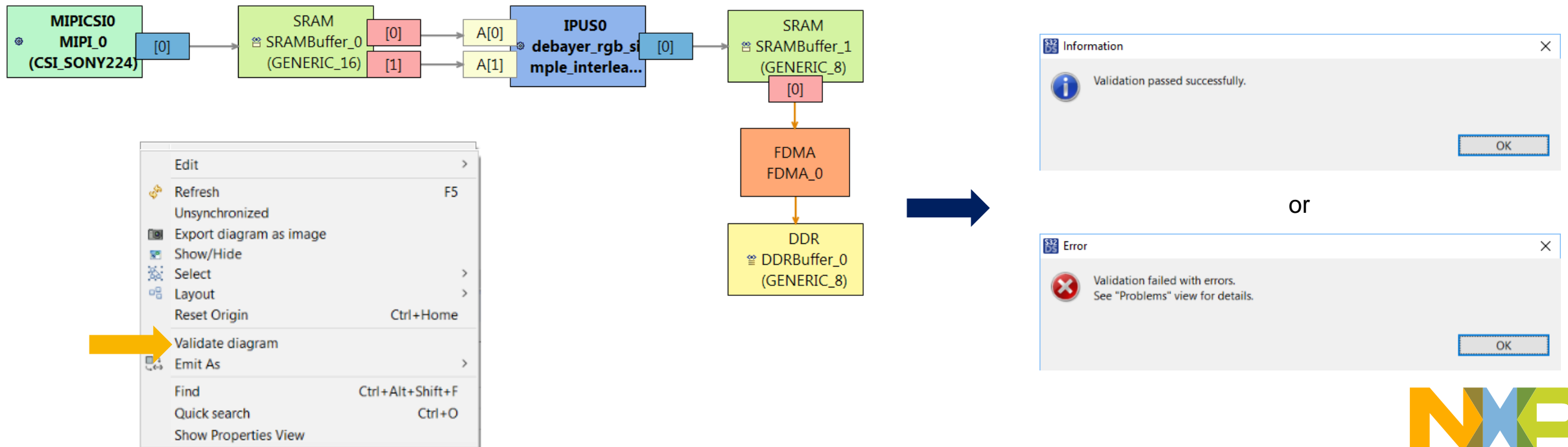
GENERATE SOURCE CODE FROM GRAPH

Once graph is constructed completely, the graph tool allows us to autogenerate source code from it. In this part, we will generate ISP source code for Linux application



Validate graph for correctness

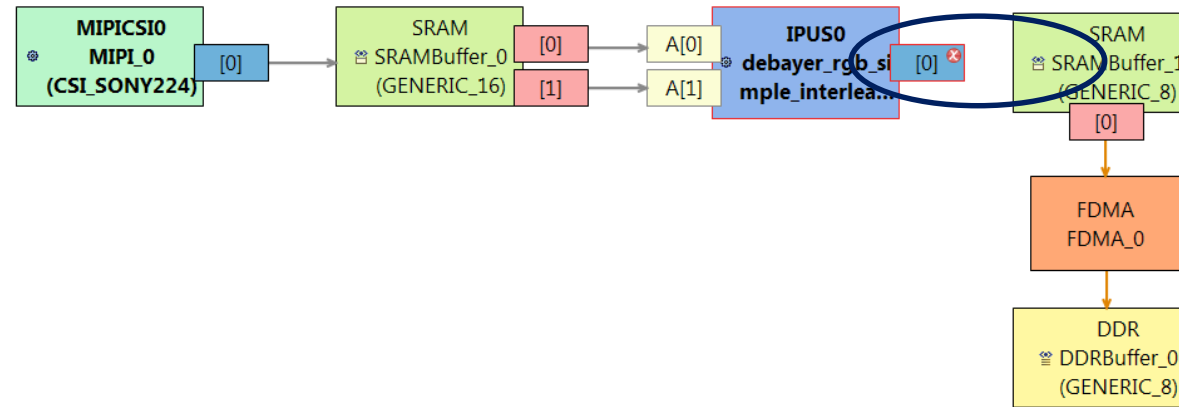
- **Save** the graph
- **Right Click** anywhere in the white part of the graph
- **Validate** graph
 - You will see a pop-up window showing status of validation.



Validation Error

1 of 2

- Error will be indicated by red cross on the block and description can be seen in the **Problems View**



Problems

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
The 'IspPortOut must have reference to IspBuffer' constraint is violated on 'IspPortOut'	ISP_VGT_test_graph.aird	/ISP_VGT_test_graph/model	ISP_VGT_test_graph::debaye...	Sirius diagram editor Plugin problems

- Find the root cause of the error(s), correct it and **Validate** your graph.

- Data type not matching

- ✘ The 'data_type_must_be_the_same' constraint is violated on '<ISPGraphClass>

- The data type between an input port of the IPU and the SRAM buffer is different

- Name error

- ✘ The 'name_has_to_start_with_alphabet_letter' constraint is violated on '

- DDR buffer configuration

- ✘ The 'stride_must_be_multiple_of_32' constraint is violated on

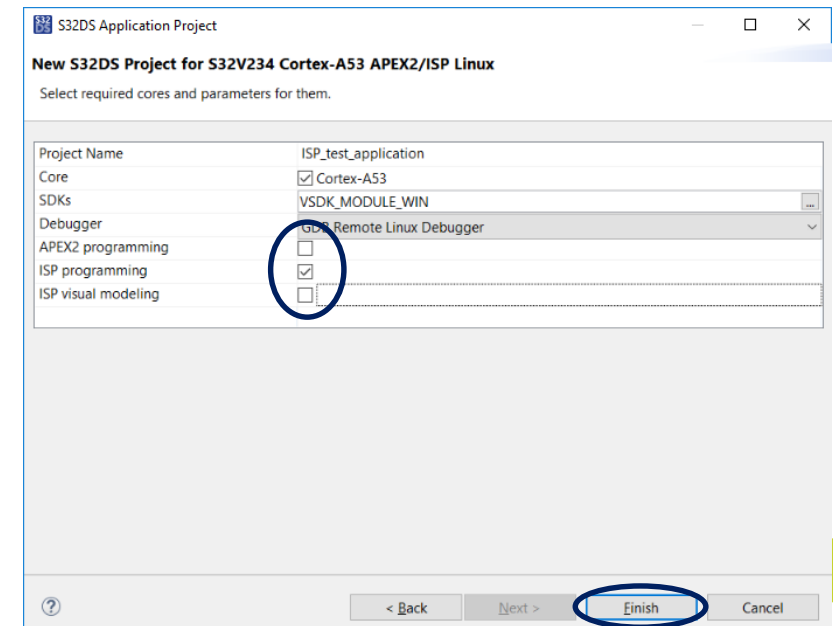
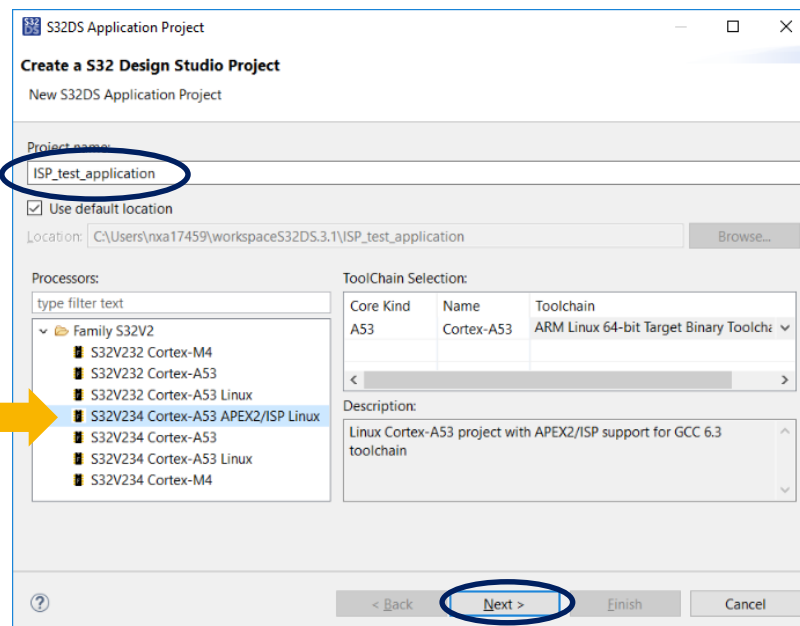
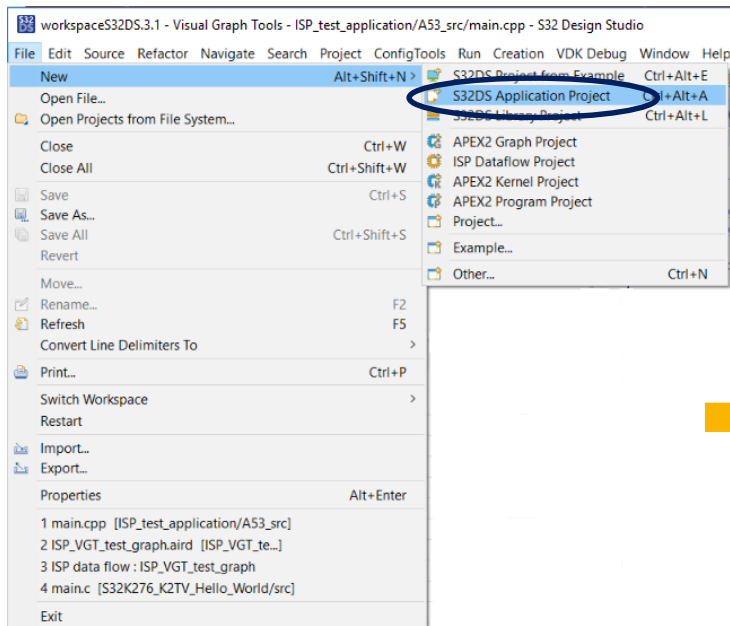
Make a Linux application project *without* an ISP graph

- Once graph is validated, next step is to generate source code from graph
- We will generate ISP code directly in this Linux application project
- So the next step is:
 - Make a new **application project** named : **ISP_test_application**

1. Go to **File** → **New** → **S32DS Application Project**

2. Type the **project name**:
ISP_test_application
3. Select **project type** as shown
4. Hit **Next**

2. Since we are developing separate ISP graph project and not using APEX, **deselect unnecessary options** as shown
3. Hit **Finish**

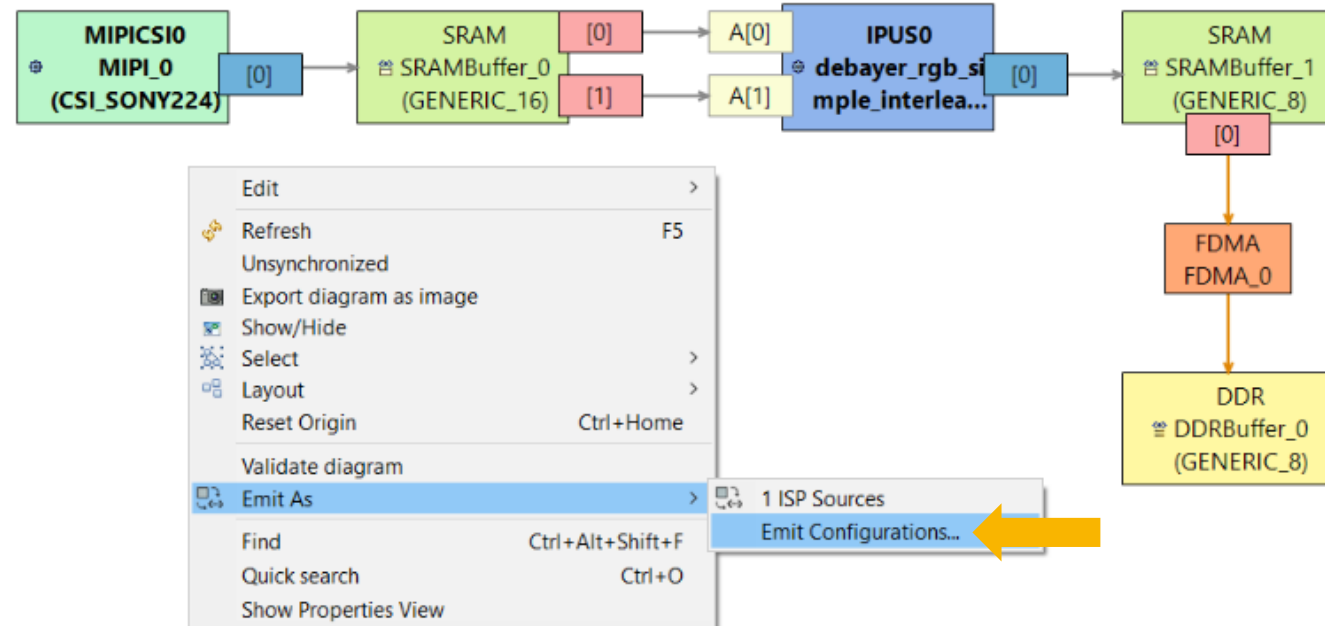


Select the destination of autogenerated source code

1 of 3

- By default all source code will be generated inside the ISP dataflow project itself
- We can reconfigure the destination of source code to any other open projects.
 - We will use this feature and generate the source code in Linux application project.

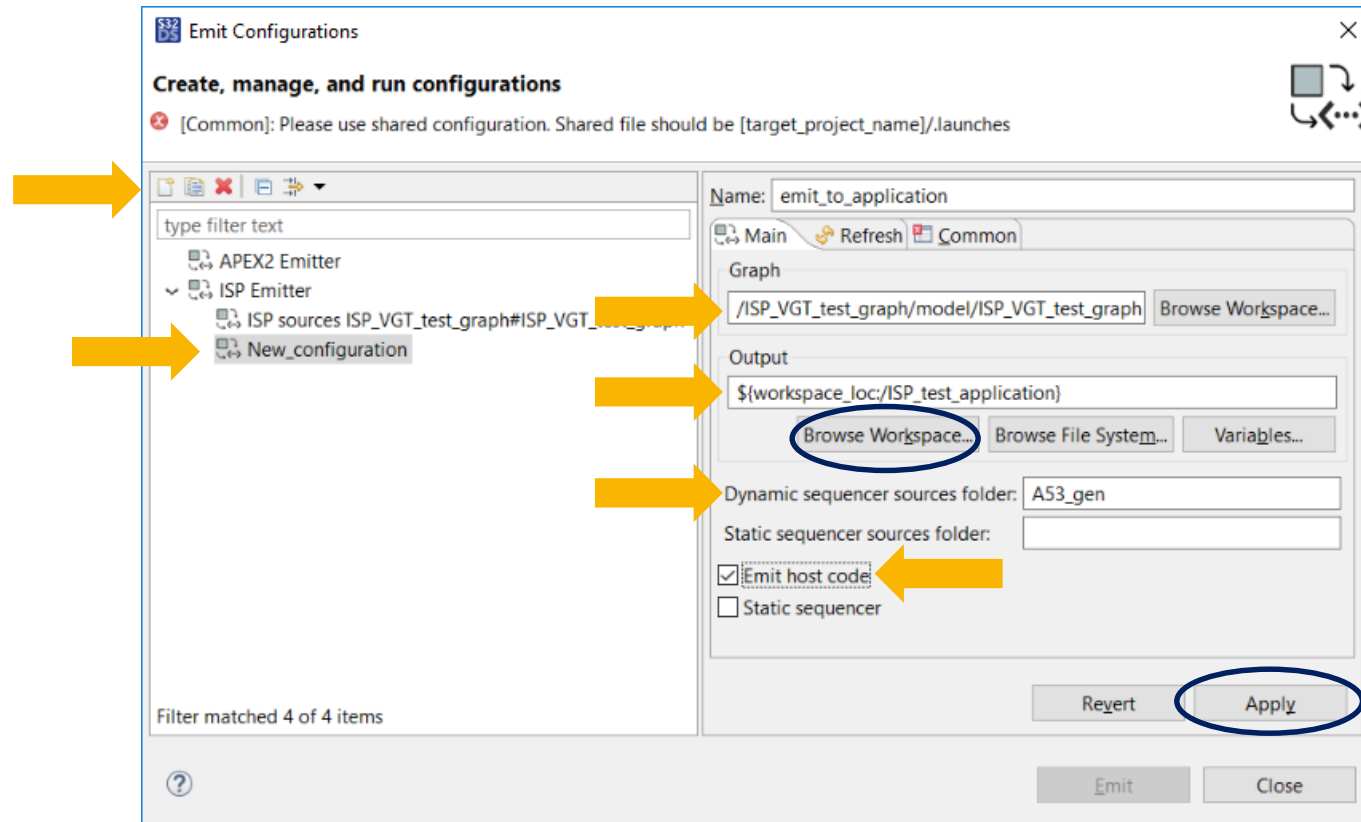
1. Select the **Emit Configuration..** option.



Select the destination of autogenerated source code

2 of 3

- Define a new configuration and specify where we want to generate our source code.
 2. **Create new configuration** as shown in the picture
 3. Click on **Apply** to save the changes

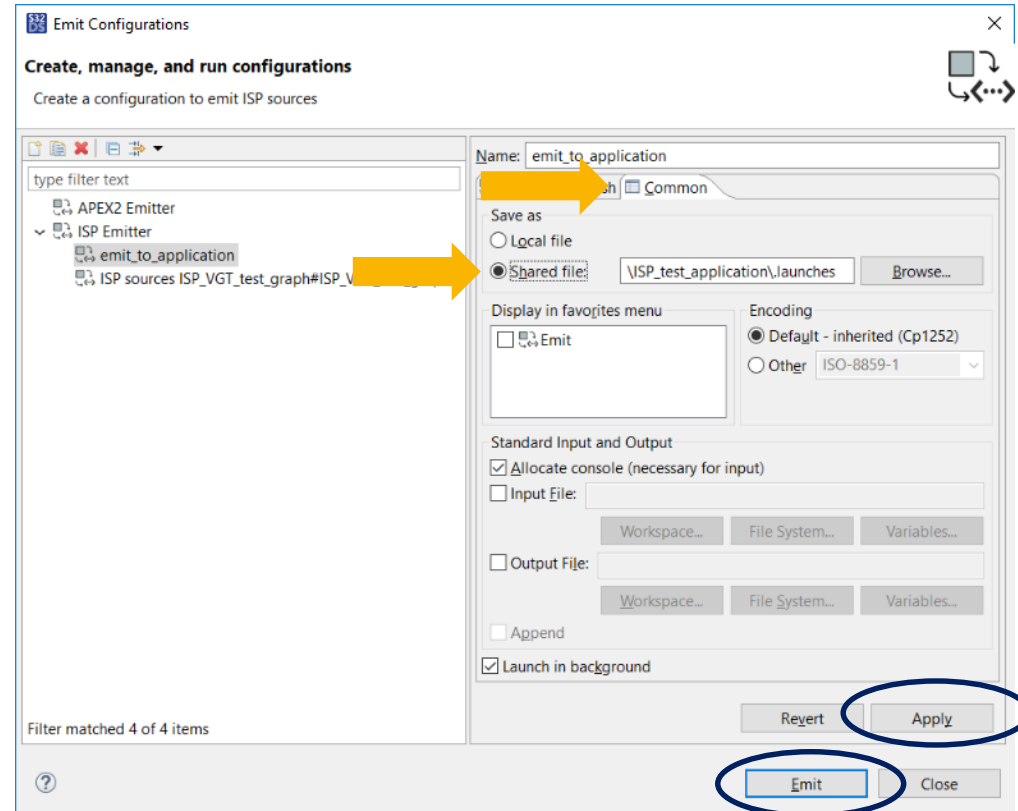


Select the destination of autogenerated source code

3 of 3

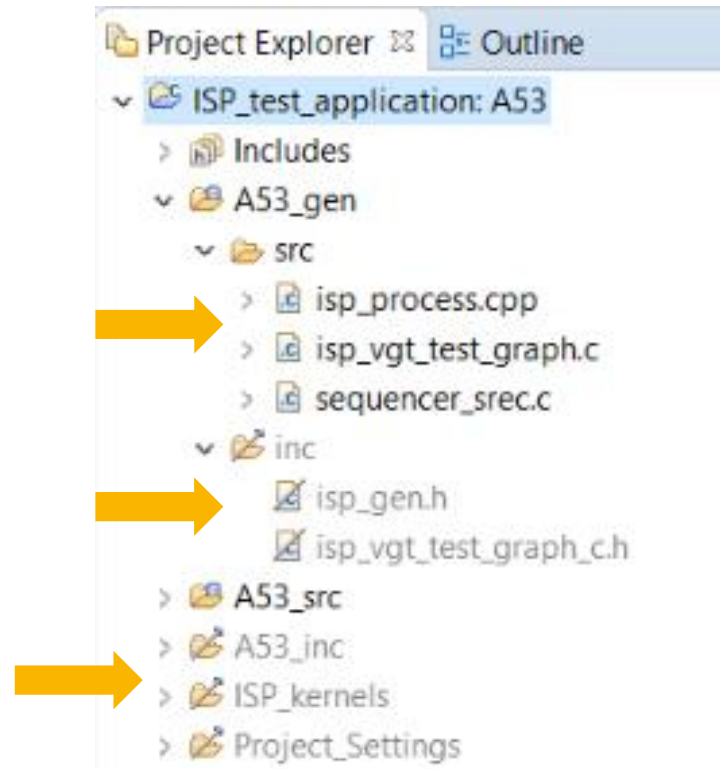
- Edit some more configuration

2. Go to **Common** tab.
3. Select the **\ISP_test_application\launches** folder under “*Shared files*” option here.
4. **Apply** the settings and Hit **Emit** button to generate a source code at the designated location



Emit the source code

- Auto generated code can be seen inside the project folder
 - Note: If you can not see source code, please right click on the project and click on **Refresh** from the menu.



LINUX APPLICATION PROJECT FOR ISP

*Now, we have everything to build an application.
Let's start building Linux application.*



Application Code for ISP

- Basic, auto generated, application code template for ISP can be found in **isp_process.cpp** in the function **ISP_CALL()**
- **ISP_CALL()** inside the **main.cpp** is just a place holder.
- User should move/add/change code inside the **isp_process.cpp** and **main.cpp** according to his/her application needs or structure
 - **Note:** In this tutorial we will not change default structure as it is not necessary

Application code for ISP : Compile

- We need to make changes & add code into application according to our requirement
 1. Our image is RGB888 type and by default DCU is configured to take YCbCr422 format. Hence, modify **A53_inc/isp_user_define.h** with following...

```
17 #define DCU_BPP DCU_BPP_24
```

2. Modify **main.cpp** to define DDR buffers that stores images coming from ISP.
 - Do not forget to add a header file: **#include "isp_vgt_test_graph_c.h"**

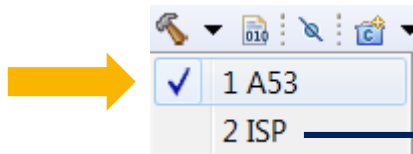
```
void io_config(sdi_grabber *lpGrabber)
{
    /* Insert the code to initialize DDR buffers */

    // *** prepare IOs ***
    sdi_FdmaIO *lpFdma = (sdi_FdmaIO*)lpGrabber->IoGet(SEQ_OTHRIX_FDMA);

    // modify DDR frame geometry to fit display output
    SDI_ImageDescriptor lFrmDesc = SDI_ImageDescriptor(WIDTH_DDR, HEIGHT_DDR, RGB888);
    lpFdma->DdrBufferDescSet(FDMA_IX_FDMA_0, lFrmDesc);

    /*** allocate DDR buffers ***/
    lpFdma->DdrBuffersAlloc(FDMA_IX_FDMA_0, DDR_OUT_BUFFER_CNT);
}
```

- Go to **C/C++ perspective** and **compile** the application for **A53** core



Info: ISP option generates binary for KRAM

Application Code for ISP: Run

Execute your `ISP_test_application.elf` binary on the target!

Connect and Observe

- Do not forget to connect Sony camera to MIPI-A port and HDMI output to display
- Run the application
- You can see camera captures streaming on the screen
 - If no output image is shown and program exits instead of continuous loop, user should check that all settings were correctly entered in the graph diagram blocks

Tips

- Don't forget to **save** and **re-generate** the source code when you change your graph
- Emit source code step validates graph first then generates source code. So, graph validation is an optional step.



SECURE CONNECTIONS
FOR A SMARTER WORLD