SDI Software User Guide

ABSTRACT:
This is the Software User Guide Document for SDI (Sensor Data Interface) library.
KEYWORDS:
User Guide, ISP, UMat
APPROVED:

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	05-May-16	Tomas Babinec	Document creation.
0.2	10-August-16	Tomas Babinec	Updated based on review findings.
1.0	07-June-17	Tomas Babinec	Updated for VSDK RTM release.
1.1	16-March-18	Nguyen Tri Hai	Updated for VSDK RTM 1.1.0 release.

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this document is to describe the Sensor Data Interface (SDI) SW library user API and is intended to serve as a reference source for VSDK based application development. For exact definitions and implementation details please refer to [1].

1.2 Scope and Objective

This document includes User Guide in the scope of the s32v234 project.

1.3 Audience Description

This document is intended for s32v234 Vision SDK users.

1.4 References

Id	*Title*	*Location*
[1]	*SDI source code documentation*	*Doxygen style comments inside the SDI source code.*
[2]	*FDMA Driver User Guide*	VisionSDK *folder:* s32v234_sdk\docs\drivers
[3]	*Sequencer Driver User Guide*	VisionSDK *folder:* s32v234_sdk\docs\drivers
[4]	*VisionSDK OAL API Specification*	VisionSDK *folder:* s32v234_sdk\docs\vsdk

Table 1: References

1.5 Definitions, Acronyms and Abbreviations

Term/Acronym	*Description*
SW	*Software*
SDK	*System Development Kit*
SDI	*Sensor Data Interface*
OpenCv	*Open library of computer vision algorithms (originated by Intel)*
OCV	*OpenCv abbreviation*
OAL	*Operating system Abstraction Layer*
IPC	*InterProcess Communication*

CMA	*Contiguous Memory Allocation*
ROI	*Region Of Interest*
GHS	*Greenhills*

Table 2: Acronyms

1.6 Document Location

VisionSDK: s32v234_sdk/docs/drivers

2 High Level Overview

The **S**ensor **D**evice **I**nterface is a host level runtime library for controlling image data input. The SDI is designed to abstract handling of the Image Signal Preprocessing (ISP) subsystem that is an integral part of the S32V234 SoC.

The ISP supports various interfaces and other HW blocks (in general referred to as ISP engines) for image sensor data

- Input: Mipi-Csi2, Viu, Ethernet, FastDMA (FDMA);
- Preprocessing: scalar/vector Image Proccessing Units (IPU), H264 decoder, Jpeg decoder, Vision Sequencer;
- Output: H264 encoder, Ethernet, FDMA.

The ISP is expected to work with high bandwidth raw sensor data and image line based granularity. To enable such preprocessing a unique Static RAM (SRAM) block has been designed. To minimize the host CPU load the ISP subsystem includes an Arm M0+ based Vision Sequencer HW block that is responsible for managing the line based processing steps, mostly handling events from various ISP engines. Thanks to these properties the ISP provides a very low latency highly programmable image data preprocessing services that come with a negligible load to host CPU.

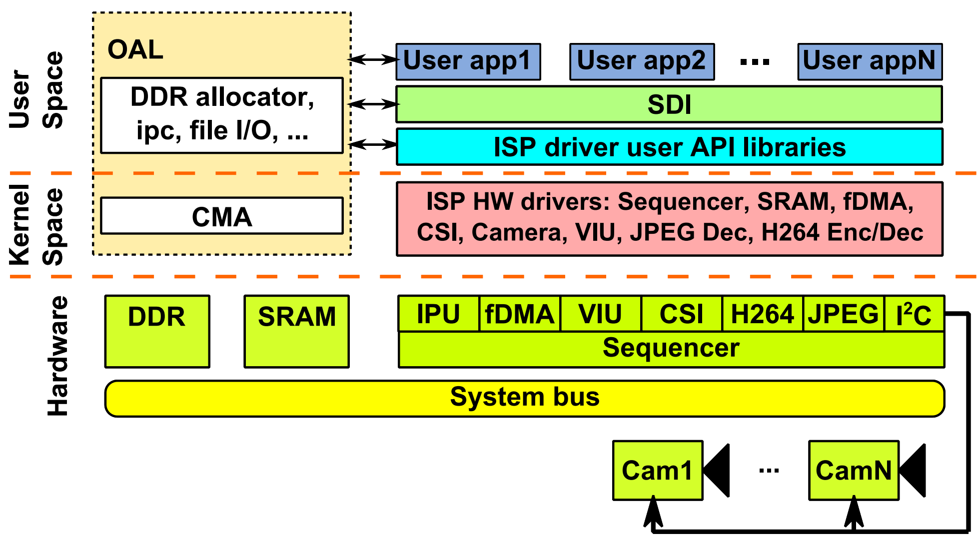

Figure 1: SDI position and interactions in s32v234 SW architecture

Figure 1 depicts the position of the SDI in the VSDK SW environment. The SDI functionality is designed mainly to cover the following:

- HW resource (ISP engine, SRAM/DDR memory) allocation;
- Configuration of the data preprocessing pipeline;

- Frame capture control.

2.1 Software Environment Interaction

As it is depicted in Figure 1, the SDI library is intended to serve as a “middleware” creating the API for user applications. To control the various HW blocks the SDI uses API provided by the drivers. To minimize OS code dependency the SDI utilizes API from the OAL (Operating system Abstraction Layer) library which includes mainly SRAM and DDR allocation services. High OCV compatibility is ensured by internal usage of UMat, which is 1:1 clone of cv::UMat data container.

2.1.1OAL library

To bypass eventual OS differences the Operating system Abstraction Layer (OAL) library is used to wrap particular OS dependent functionality. This includes memory allocation, file I/O, management of threads, processes, IPC (Inter Process Communication) tools, etc.

Part of the OAL is a kernel module, which is used for allocation of physically contiguous memory that is not managed by the OS. A physically contiguous memory allocation is required for processing steps done by various HW accelerators including ISP and APEX. At the same time the OAL allocated blocks can be utilized for creation of data regions that are shared among separate address spaces.

2.1.2OpenCv compatibility

For comfortable image data storage and management, the SDI library adopts UMat image data container for internal representation of frame buffers. Since the vsdk::UMat is 1:1 clone of the cv::UMat, the input and output data from UMat can be directly used as OCV functions parameters.

3 Functional Description

The aim of this section is to give detailed description of the SDI library features and their internal implementation with respect to currently supported functionality. Please refer to the Doxygen generated documentation [1] for exact definition of SDI infrastructure and API.

3.1 Internal Structure

From the binary point of view the SDI library build generates a `sdilib.a` file, which has to be linked to every application that is using the SDI.

From the source code perspective the SDI has three files/modules `sdi_graph.cpp`, `sdi_io.cpp` and `sdi.cpp`. (see Table 3).

Module file	***Short description***
`sdi.cpp`	Provides infrastructure for ISP resource sharing among threads and processes. Implements high-level API for ISP graph use.
`sdi_io.cpp`	Introduces Input/Output objects for configuration of various ISP engine that can be used in the ISP graph. Internally uses the functionality from `sdi_graph.cpp`.
`sdi_graph.cpp`	Implements ISP graph handling.

Table 3: SDI source code modules

The SDI environment is in each application represented by a static `sdi` class instance that is shared by all potential threads the application might be divided to.

There are several global variables that are used to keep current status of the SDI library. Each process that uses `sdilib.a` has its own copy of these variables but they are shared among threads in the same address space. If an access to the global variables is required during SDI development the multi process/thread safety has to be maintained.

User applications do not have direct access to SDI global variables. As a result there are no special demands on the SDI users except the proper use of `sdi::Initialize()` and `sdi::Close()` calls.

The SDI internal process/thread safety mechanisms are disabled. It is strongly advised to operate SDI functionality from single process/thread.

3.2 SDI Lifecycle

To signal beginning and end of the SDI library usage, each client application (each process/thread that utilizes SDI services) has to call `sdi::Initialize()` and `sdi:Close()` functions.

3.3 Sensor Data Processing

One of the main objective of the SDI library is to enable user application with access to sources of image data. This usually requires:

- Facilitation of sensor hardware parameters set up (frames per second, exposure, etc.);
- Image data grabbing
  - data transfer from sensor to user specified memory address;
  - data preprocessing (gamma correction, exposure control, etc.);

To cover all the above mentioned functionality the SDI implements several data types and methods described in the following 3 subsections. A simple use case example of SDI library functionality is introduced in section **Error! Reference source not found.**.

3.3.1 Class sdi_graph

Class that encapsulates ISP graph data structure related functionality. The ISP graph [3] defines the image sensor data preprocessing that is executed by the ISP subsystem. The graph can be designed in a graphical way using an S32 Design Studio for Vision (S32DS). The S32DS generates to a C language code. The resulting *.c file contains mainly the following:

- `gpGraph`: representation of the ISP graph as an array of pointers to `SEQ_Head_t` structures. The contents of this structure have to be finalized by the SDI and downloaded to the M0 memory to serve as a runtime guideline for the Sequencer FW when managing the ISP subsystem HW.
- `gGraphMetadata`: additional information that was specified during the graph design time and is required by the SDI to configure the preprocessing pipeline. This includes information about the ISP graph array itself as well as number of sensor their types and other important configuration.

Both the `gpGraph` and `gGraphMetadata` have to be passed to the `sdi_graph` object constructor. The `sdi_graph` constructor first copies all ISP graph components to one contiguous memory block and parses the structure to get shortcuts to important objects like pointers to the data input/output nodes (e.g.: CSI, VIU, JPEG, FDMA, …).

When `sdi_graph` was constructed successfully the user application is expected to apply graph configuration updates if required (e.g. FDMA transfer descriptors) and most of all to provide the addresses of allocated physically contiguous DDR buffers.

After the graph configuration was updated from the user application the `Finalize()` method of the `sdi_graph` can be called to allocate required SRAM buffer and FDMA transfer channels. If successful the graph can be downloaded to the Sequencer memory by invoking the `Download()` method.

Full listing of the `sdi_graph` methods with short description can be found in Doxygen documentation [1] generated from the SDI source code.

3.3.2Class sdi_io

The sdi_io creates a utility SW layer above the sdi_graph class API. The sdi_io is an abstract base class that declares mandatory properties and functionality of objects that represent input or output nodes (ISP engines) in the ISP graph. The main objective of the "io" layer is to unify the handling of the various ISP engines that are included in the ISP subsystem.

Each sdi_io derived object has to support the following methods:

- `Reserve()`

  Establishes exclusive access to the related HW blocks within the scope of the SDI environment. Usual implementation includes HW driver initialization and access locking.

- `Release()`

  Cancels the previously established exclusive access to the related HW blocks. Usual implementation includes HW driver cleanup, close and access unlocking.

- `Setup()`

  Applies previously specified configuration to the actual HW block.

  Enabled after successful `Reserve()` call.

- `Start()`

  Puts the related HW block into an operational state. E.g.: enables Camera data transmission on a CSI interface.

- `Stop()`

  Terminates the related HW block operation. E.g.: disables Camera data transmission on a CSI interface.

The sdi_io derived objects that are supported in the VSDK are listed below:

- `sdi_FdmaIO`

  Abstracts the configuration of FDMA engine and assigned DDR buffers handling.

  Each ISP graph can utilize up to 16 FDMA channels for data transfers between SRAM and DDR. The SRAM buffers are considered to be part of the graph and have to be specified during the graph design time. On the other hand the DDR buffers are understood as external resource that can be influenced by the user application.

  This means that the user has the freedom to use the default DDR buffer configuration or to choose a different buffer geometry. This feature allows to implement many useful operation (e.g. ROI) without any added computation load to the host CPU.

  The FDMA channels are described by a `FDMA_Tc_t` structure. For more information please refer to [4]. During the graph runtime the FDMA channels are continuously used to compose full frames in the DDR memory. Because of this at least 2 independent DDR buffers should be provided for each active FDMA channel to rule out any race conditions while the application is reading data from the a buffer that is at same time being written by the ISP graph or vice versa. For more information about DDR buffer handling please refer to Sequencer driver documentation [5].

The used DDR buffers can be either allocated by one of the `DdrBufferAlloc()` methods that are introduced by the `sdi_FdmaIO` or provided to the object from user application through call to `DdrBufferSet()` method. For more advanced FDMA configuration the `TcGet/Set()` methods have been implemented. Please refer to [1] for exact definitions.

- `sdi_H264EncIO`

  Abstracts the configuration of H264 encoder engine.

- `sdi_JpegDecIO`

  Abstracts the configuration of JPEG decoder engine.

  Currently supports single stream configuration only.

- `sdi_MipiCsiIO`

  Abstracts the configuration of MipiCsi2 receiver engine. Includes also the setup of related sensor devices. The sensor device info is automatically gathered from ISP graph metadata. So far the following MipiCsi connected cameras are supported:

  - Sony IMX224.
  - Maxim Serializer/Deserializer HW setup with 4 Omnivision Ov10640 cameras.
  - Maxim Serializer/Deserializer HW setup with 4 Omnivision Ov10635 cameras.
  - Maxim Serializer/Deserializer HW setup with 4 Sony IMX224.
  - TIUB964 camera.
  - Omnivision Ov10640 camera.
  - Omnivision Ov10635 camera.

  Because each application can have slightly different demands on the sensor configuration, the `sdi_MipiCsiIO` object provides only the basic setup. It is up to the user application to implement any specific configuration steps based on the particular HW driver capabilities.

- `sdi_ViuIO`

  Abstracts the configuration of Viu receiver engine. Includes also the setup of related sensor devices. The sensor device info is automatically gathered from ISP graph metadata. So far the following Viu connected cameras are supported:

  - Omnivision Ov10635.
  - Omnivision Ov10640.

  Because each application can have slightly different demands on the sensor configuration, the `sdi_ViuIO` object provides only the basic setup. It is up to the user application to implement any specific configuration steps based on the particular HW driver capabilities.

The `sdi_io` SW layer implements also two utility objects:

- `SDI_Frame`

Incorporates an `UMat` image container with SDI related metadata (FDMA channel index, and index of the particular buffer among the DDR buffers assigned the particular FDMA channel).

- `SDI_DdrBufferArr`

  Encapsulates handling of a set of DDR buffers assigned to one FDMA channel. This includes physically contiguous memory allocation and also region of interest (ROI) definition.

3.3.3Class sdi_process

The `sdi_process` creates a utility SW layer above the `sdi_graph` class API. In the current state of implementation the `sdi_process` only mirrors the top-level ISP graph management API available already in the `sdi_graph` object (the `Finalize()`, `Download()` methods).

The main objective of the `sdi_process` layer is to provide a possibility for future expansion of the SDI library to multi graph support and non-Sequencer managed ISP preprocessing.

3.3.4Class sdi_grabber

The `sdi_grabber` class serves as a top-level ISP preprocessing management object. Internally the `sdi_grabber` instance encapsulates sdi_proccess defining the ISP data preprocessing pipeline as well as the `sdi_io` derived objects for input/output ISP engines configuration. The actual data grabbing sequence is managed by `sdi_grabber` class instances. It covers the whole data path from sensor device to user specified memory buffer.

The usual lifecycle of the `sdi_grabber` has the following stages:

1. Creation a `sdi_grabber` instance:

```
sdi_grabber *lpGrabber = new(sdi_grabber);
sdi::Initialize(0);  //initialize SDI environment
```

2. Specificaion of the ISP graph preprocessing pipeline:

```
lpGrabber->ProcessSet(gpGraph, &gGraphMetadata);
```

3. **Optional:** Installation of Sequencer event callback (see [5] for possible event listing):

```
lpGrabber->ProcessSet(gpGraph, &gGraphMetadata);
```

4. Preparation of sdi_io objects. In most cases at least FDMA is present:

```
sdi_FdmaIO *lpFdma = (sdi_FdmaIO*)lpGrabber->IoGet(SEQ_OTHRIX_FDMA);
// allocate DDR_BUFFER_CNT buffers for FDMA trahnsfer channel 0
lpFdma->DdrBuffersAlloc(0, DDR_BUFFER_CNT);
```

5. Reserve HW resources & preconfigure the HW blocks:

```
lpGrabber->PreStart();
```

6. Additional HW configuration:

```
// Camera Configuration
// modify camera geometry setup before setting up expsoure control
SONY_Geometry_t lGeo;
SONY_GeometryGet(CSI_IDX_0, &lGeo); // get current setup
lGeo.mVerFlip = 1; //apply vertical flip of the image
lGeo.mHorFlip = 1; //apply horizontal flip of the image
lGeo.mFps = 15;    //reduce frame rate to 15fps
SONY_GeometrySet(CSI_IDX_0,&lGeo);
```

7. Initiate the ISP pipeline:

```
lpGrabber->Start();
```

8. In an endless loop use the preprocessed frames:

```
for(;;)
{
    lFrame = lpGrabber->FramePop();
    if(lFrame.mImage.mData == NULL)
    {
      printf("Failed to grab image number %u\n", lFrmCnt);
      break;
    } // if pop failed

    //<user defined processing>

    if(lpGrabber->FramePush(lFrame) != LIB_SUCCESS)
    {
      break;
    } // if push failed
    lFrmCnt++;
}
```

9. Once the processing has ended cleanup resources:

```
//*** Stop ISP processing ***
lpGrabber->Stop();

// clean up grabber resources
lpGrabber->Release();

delete(lpGrabber);

sdi::Close(0);  // exit SDI environment
```

The above-mentioned stages can be used to generate a skeleton SDI frame input based application.

3.4 Image Containers

To store and work with image data SDI library uses the `SDI_Frame` which is described at the end of section 3.3.2. As it was mentioned there the `SDI_Frame` structure incorporates a `UMat` image container class member.

As mentioned before the UMat type is fully compatible with cv::UMat class. Internally the vsdk::UMat is utilizing OAL library services to ensure contiguous memory regions allocation which are required by ISP, APEX and some other s32v234 subsystems.

`UMat` is only a representation of the physical memory region. To be able to directly access the pixel data the user is expected to call the `UMat::getMat()` method which return an instance of `vsdk::Mat`. The `vsdk::Mat` among other things contains a smart pointer to virtual mapping of the physical memory region represented by `UMat`.

For more information about VSDK `UMat` and `Mat` classes, please refer to corresponding documents and OCV documentation.

3.5 Platform Differences

The SDI library functionality is currently available for the following platforms: s32v234.

4 Use Case Examples

This chapter discusses several SDI related use-case examples that can be also found in the VSDK demos.

An example of a SDI frame input based skeleton application was already introduced in chapter 3.3.4. The same scheme is depicted in Figure 2.

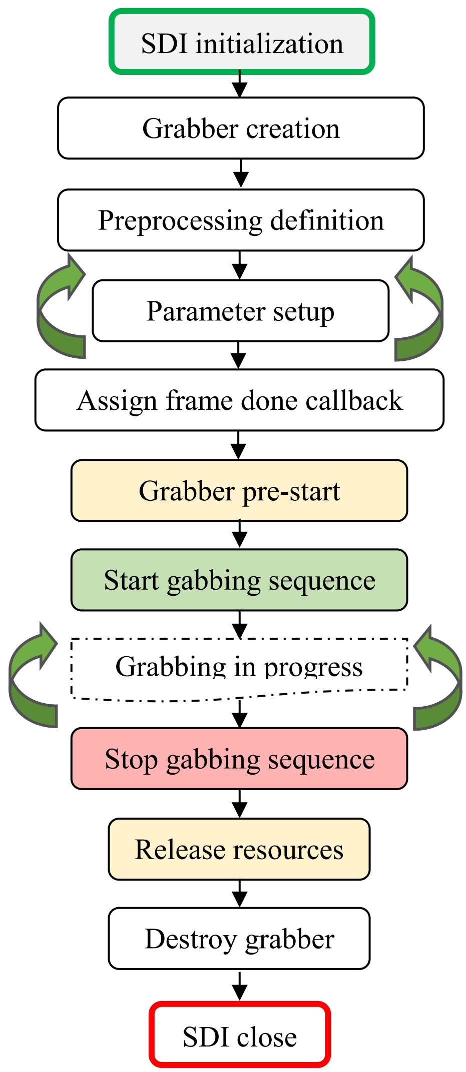

Figure 2: SDI based frame input skeleton application diagram

4.1 Merging SRAM buffers to one DDR frame

The following chapter gives an example of how to use the FDMA engine to merge several SRAM buffers into one DDR frame.

The following example is taken from the `isp_yuv_grey_pyramid` demo.

The Figure 3 depicts the subsection of a `yuv_grey_graph` graph where two FDMA channels (named Y2_SCALED_TO_DDR and Y4_SCALED_TO_DDR) are being used to transfer two levels of grayscale image pyramid from SRAM buffers to DDR memory.

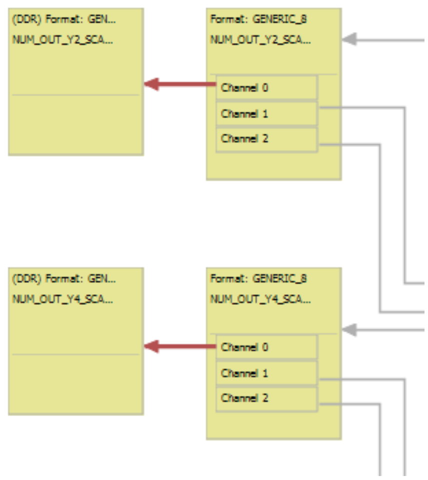

Figure 3: yuv_grey_pyramid graph subusection

The demo application uses the FDMA configuration to put all the gray pyramid levels as ROI into one DDR buffer. To achieve this, the following steps were required:

1. Request of `sdi_FdmaIO` object:

```
sdi_FdmaIO *lpFdma = (sdi_FdmaIO*)lpGrabber->IoGet(SEQ_OTHRIX_FDMA);
```

2. Generate array of grayscale DDR buffers with full frame size (1280x720):

```
#define IMG_WIDTH  1280
#define IMG_HEIGHT 720
// *** Gray pyramid ***
// *** HD Y frame ***
SDI_ImageDescriptor lGrayFrmDesc;
SDI_DdrBufferArr lGrayBuffArr;
lGrayFrmDesc = SDI_ImageDescriptor
(
  IMG_WIDTH,
  IMG_HEIGHT,
  GS8         // grayscale image 1byte per pixel (stride == IMG_WIDHT)
);
// allocate contiguous memory for 3 buffers
lGrayBuffArr.Allocate(lGrayFrmDesc, DDR_OUT_BUFFER_CNT);
```

3. Create a ROI from SDI_DdrBufferArr for first FDMA channel (first level of the pyramid):

```
// *** Y2_Scaled channel ***
SDI_DdrBufferArr  lGrayBuffArrRoi;
lGrayBuffArrRoi = lGrayBuffArr.Roi  //create a ROI based on HD Y buffer array
(
  0,                                 // x coordinate of top left ROI corner
  0,                                 // y coordinate of top left ROI corner
  NUM_OUT_Y2_SCALED_BYTES,          // width of the roi in pixels
  NUM_OUT_Y2_SCALED_LINES           // height of the roi in pixels
);
// modify FDMA channel index this ROI belongs to
lGrayBuffArrRoi.ChannelIdxSet(FDMA_IX_Y2_SCALED_TO_DDR);
// register the ROI to the FDMA channel index
lpFdma->DdrBuffersSet(lGrayBuffArrRoi);
```

4. Create a ROI from SDI_DdrBufferArr for second FDMA channel (second level of the pyramid). Y coordinate is now different:

```
// *** Y4_Scaled channel ***
lGrayBuffArrRoi = lGrayBuffArr.Roi  //create a ROI based on HD Y buffer array
(
  0,                                 // x coordinate of top left ROI corner
  NUM_OUT_Y2_SCALED_LINES,           // y coordinate of top left ROI corner
  NUM_OUT_Y4_SCALED_BYTES,          // width of the roi in pixels
  NUM_OUT_Y4_SCALED_LINES           // height of the roi in pixels
);
// modify FDMA channel index this ROI belongs to
lGrayBuffArrRoi.ChannelIdxSet(FDMA_IX_Y4_SCALED_TO_DDR);
// register the ROI to the FDMA channel index
lpFdma->DdrBuffersSet(lGrayBuffArrRoi);
```

5. Repeat the same approach for remaining pyramid levels.

4.2 Dividing one SRAM buffer into two DDR frames

The following chapter gives an example of how to use the FDMA engine to separate SRAM buffer that contains two images side-by-side.

The following example is taken from the `isp_stereo_apexbm` demo.

The Figure 4 depicts the subsection of a `isp_stereo_ftf` graph where two FDMA channels (named FastDMA_Right_Out and FastDMA_Left_Out) are being used to transfer left and right image parts into separate DDR buffers.

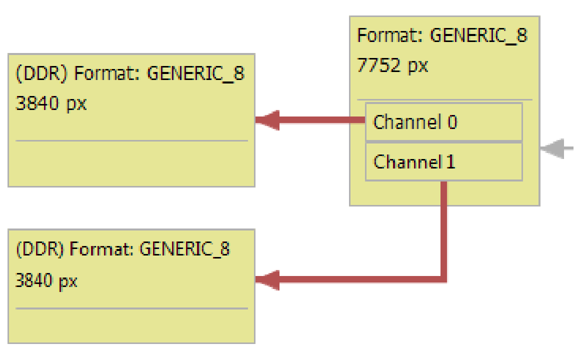

Figure 4: yuv_grey_pyramid graph subusection

The demo application uses the FDMA configuration to divide the image data into two distinct DDR buffers. To achieve this, the following steps were required:

1. Request of `sdi_FdmaIO` object:

```
sdi_FdmaIO *lpFdma = (sdi_FdmaIO*)lpGrabber->IoGet(SEQ_OTHRIX_FDMA);
```

2. Setup identical geometry for both left and right DDR buffers with full frame size:

```
#define SCR_WIDTH  1280  //width of the screen
#define SCR_HEIGHT 720
#define SRC_WIDTH  1292  //width of one (left or right) source frame
// *** RGB888 images ***
SDI_ImageDescriptor lFrmDesc = SDI_ImageDescriptor(
                                 SCR_WIDTH,
                                 SCR_HEIGHT,
                                 RGB888); //24 bpp (stride = SCR_WIDTH*3)
//both left/right images have the same descriptor
lpFdma->DdrBufferDescSet(FDMA_IX_FastDMA_Right_Out,  lFrmDesc);
lpFdma->DdrBufferDescSet(FDMA_IX_FastDMA_Left_Out,   lFrmDesc);
```

3. Let the buffers be allocated:

```
//*** allocate DDR buffers ***
lpFdma->DdrBuffersAlloc(DDR_BUFFER_CNT);
```

4. Update SRAM side of the FDMA transfer descriptors:

```
FDMA_Tc_t lTc; // FDMA transfer channel structure

// right: update transfer size
lTc.mTdm.mTdIdx = FDMA_IX_FastDMA_Right_Out; //set channel index in metadata
lpFdma->TcGet(lTc);                          //fetch current TD config
lTc.mTd.mLineSize = (SCR_WIDTH) * (uint32_t)io::IO_DATA_CH3;
lpFdma->TcSet(lTc);                          //set updated TD config

// left: shift SRAM address of the FDMA transfer and update transfer size
lTc.mTdm.mTdIdx = FDMA_IX_FastDMA_Left_Out;
lpFdma->TcGet(lTc);
lTc.mTd.mSramImgStartAddr += (SRC_WIDTH) * (uint32_t)io::IO_DATA_CH3;
lTc.mTd.mLineSize          = (SCR_WIDTH) * (uint32_t)io::IO_DATA_CH3;
lpFdma->TcSet(lTc);
```