Frame Input ISP Software User Guide

ABSTRACT:
This is the Software User Guide Document applicable for Frame Input ISP library.

KEYWORDS:
SRAM, allocator, Linux, driver
APPROVED:

AUTHOR	SIGN-OFF SIGNATURE #1	SIGN-OFF SIGNATURE #2
Tomas Babinec		

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.1	04-May-16	Tomas Babinec	Initial version
0.2	10-August-16	Tomas Babinec	Updated based on review

Table of Contents

LIST OF TABLES

1 Introduction

1.1 Purpose

The purpose of this document is to define the API of the FrameInputISP object from frame_io library. For exact definitions and implementation details please refer to [1].

1.2 Scope and Objective

This document gives a high-level understanding of the mechanisms available to configure and use the ISP preprocessing pipeline from the user applications including the access to preprocessed frames in DDR.

1.3 Audience Description

This document is intended for users of the s32v234 Vision SDK.

1.4 References

Id	*Title*	*Location*
[1]	*Frame Input ISP source code*	*VSDK release*
[2]	*SDI SW ADD*	*Vision sdk git, folder:* s32v234_sdk\docs\drivers\
[3]	*Frame_output_v234fb*	*Vision sdk git, folder:* s32v234_sdk\docs\drivers\
[4]	*Frame_output_dcu*	*Vision sdk git, folder:* s32v234_sdk\docs\drivers\

Table 1 References Table

1.5 Definitions, Acronyms, and Abbreviations

Term/Acronym	*Description*
SW	*Software*
HW	*Hardware*
IP	*Intellectual Property*
API	*Application Programming Interface*
SRAM	*Static Random Access Memory*

DRAM	*Dynamic Random Access Memory*
DDR	*Double Data Rate DRAM*
fDMA	*fast Direct Memory Access HW block*
SDI	*Sensor Data Interface library*
SoC	*System on Chip*
ISP	*Image Signal Processing subsystem*

Table 2 Acronyms Table

1.6 Document Location

This document is available at the following location: vision SDK Git repository (ssh://git@sw-stash.freescale.net/vswat/vsdk.git) in folder s32v234_sdk\docs\drivers\.

2 General Description

The FrameInputISP object is part of the frame_io library from VSDK. It has been developed to create a simple mechanism for ISP pipeline configuration and pre-processed frames' access from the user application.

The FrameInputISP has to be understood as a temporary solution, which was replaced by the SDI (Sensor Device Interface) library in VSDK 0.9.3 release.

3 Functional Description

The FrameInputISP acts as a convenience layer between the HW drivers' API and user application (see Figure 1). For ISP graph configuration and management the FrameInputISP object uses part of the SDI library. From the 0.9.3 VSDK release the FrameInputISP functionality is replaced by SDI.

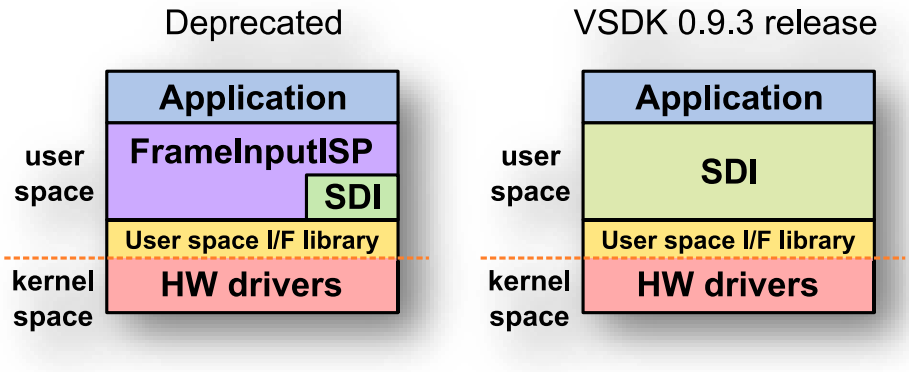

Figure 1: FrameInputISP role

3.1 Main tasks

- Graph preprocessing (using SDI graph module):
  - Allocation of resources including SRAM buffers and HW reservation.
  - Download of ISP FW (M0+ code, ISP graph and IPU kernels).
- Configuration of DDR buffer streams based on user input.
- HW control:
  - Sequencer FW boot.
  - Initial configuration of the camera.
- DDR frame buffers' handling - Get/PutFrame() functionality.

3.2 API

Method	Description
`FrameInputISP`	Constructor. Calls IspInit().
`IspInit`	Internally handles the ISP graph structure preprocessing and initial configuration of the HW. Supports CSI and VIU cameras if connected. Calls MipiCsiInit() and ViuInit.

`MipiCsiInit`	Configures MipiCsi receiver HW block based on the parameters provided in the graph.
`ViuInit`	Configures VIU receiver HW block based on the parameters provided in the graph.
`GetFrame`	Attempts to get an ISP preprocessed frame to user application. Its parameter specifies the stream index from which the frame should be taken. If no parameter was specified the frame is taken from the stream with index 0. Repeatedly calls GetFrameNonBlock() until there is a frame ready or the timeout count is reached. Returns physical address to the DDR frame if successful, otherwise NULL.
`GetFrameNonBlock`	Attempts to get an ISP preprocessed frame to user application. Its parameter specifies the stream index from which the frame should be taken. If no parameter was specified the frame is taken from the stream with index 0. Returns physical address to the DDR frame if successful, otherwise NULL.
`PutFrame`	Returns last frame acquired through GetFrame back to the ISP preprocessing pipeline.
`GraphFetch`	Returns a pointer to ISP Graph encapsulating object. It can be used for debugging purposes or to implement more complex functionality not supported by the FrameInputISP object API.
`DdrBuffersSet`	Configures specific buffer streams properties. Makes the user allocated DDR buffers available for ISP preprocessing pipeline.
`Start`	Starts the ISP graph execution. The graph will wait for first HW events to come (e.g. line_done signal from MIPICSI receiver).
`Stop`	Requests the ISP graph to be terminated gracefully.
`StartCam`	Enables the camera data input stream. This usually invokes the ISP pipeline execution.

Table 3: FrameInputISP API

3.3 Workflow

To use the FrameInputISP object the user application has to include the `frame_input_isp.h` header.

Now the ISP object can be created by the following constructor call:

```
io::FrameInputISP lIsp;.
```

The user application is responsible for allocation of the DDR buffers. The buffers have to be physically contiguous. To allocate a physically contiguous memory block the `OAL_MemoryAllocFlag()` from OAL can be used. The function returns virtual addresses. The ISP works only with physical address so the `OAL_MemoryReturnAddress()` function has to be used to get access also to the physical address of the memory.

```
io::IspBufferStream lBufferStream;

lBufferStream.mStreamIdx  = 0;
lBufferStream.mCnt        = DDR_BUFFER_CNT;
lBufferStream.mBaseShift  = 0;
lBufferStream.mLineStride = WIDTH * (uint32_t)io::IO_DATA_CH3;
lBufferStream.mLineCnt    = 720; //495;//HEIGHT;

lBufferStream.mpBuffers   = lpFbsPhysical;

// init stream 0
lIsp.DdrBuffersSet(lBufferStream);
```

To make the allocated DDR buffers available to the ISP preprocessing pipeline an `io::IspBufferStream` structure has to be filled out and passed to the FrameInputISP object using its `DdrBuffersSet()` method.

The ISP graph can generate/consume several independent DDR frame sequences (streams) at a time (e.g.: downscaled grayscale image intended for vision processing, together with RGB HD for display). To distinguish between the DDR buffer streams the `mStreamIdx` is used.

To rule out race conditions between ISP and other processing steps that access the DDR buffers, at least 2 DDR buffers for each stream have to be allocated. To reach better behavior, it is recommended to have 3 DDR buffers available. Number of available buffers has to be set to `mCnt`.

The `IspBufferStream` structure also requires the SRAM to DDR data transfer through FDMA to be configured, by setting up:

- `mBaseShift`
  byte offset to be added to each DDR address to start the data transfer from;
- `mLineStride`
  bytes per line in the DDR buffer.
- `mLineCnt`
  number of lines available in the DDR buffer.

The actual addresses of the DDR buffers are expected to be provided as an array of HW 32bit address set to the `mpBuffers` member of the `IspBufferStream` structure. As a last step in the DDR buffer setup the `FrameInputISP::DdrBuffersSet()` method has to be called.

Lifecycle of the DDR buffers provided to the ISP pipeline is depicted in Figure 2.

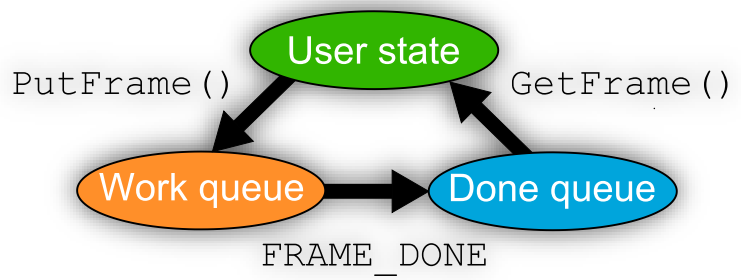

Figure 2: DDR buffer lifecycle

Each buffer can assume the following states:

- "User state" – the buffer is safe to be used by the user application until pushed back to the work queue through `FrameInputISP::PutFrame()`, or by successive call to `FrameInputISP::GetFrame()`.
- "Work queue" – the buffer is scheduled to be used by the ISP pipeline and must not be accessed from the user application. Otherwise image artefacts can be observed caused by the ISP/Host simultaneous access.
- "Done queue" – the buffer was finished by the ISP and is ready to be made available to the user application after calling `FrameInputISP::GetFrame()`.

At this point the ISP graph can be started using `FrameInputISP::Start()` method. It has to attributes which can be left unset to use the defaults:

1. Number of frames to capture. If not used or set to zero, the graph will be executed indefinitely, until the stop is requested. Otherwise just the number of frames will be provided at the output and then the graph execution will be terminated.
2. Number of input lines to be expected at the input of the ISP graph. To use the graph default value, should be left empty or set to 0. It is intended for debug purposes to override the default value from graph.

```
lIsp.Start();.
```

In most cases the ISP uses camera feed as input. So to when the graph was started it has to wait for the first camera line done signals to come. To start the camera the `FrameInputISP::StartCam()` method has to be called. Just prior to this call any camera specific configuration can happen.

```
lIsp.CamStart();.
```

Now the ISP is working independently, copying the full frame data to the DDR buffers provided in previous setup steps. An example of the grabbing loop (from isp_csi_dcu demo) is provided below. To get access to the frames finished by the ISP the `FrameInputISP::GetFrame()` method has to be called. By default it takes the frames from the stream index 0. To get the frames from different stream, its index has to be provided as a parameter to the `GetFrame()` method.

The `GetFrame()` method returns physical address of the DDR buffer. To use it under Linux the `OAL_MemoryReturnAddressPhysical()` function can be called to get the corresponding virtual mapping pointer to the buffer.

The example grabbing loop contains also the display output using `PutFrame()` method of the `io::FrameOutputDCU` or `io::FrameOutputV234Fb` object which are described in detail in a different documents [3, 4].

```
void *lpFrame = NULL;
uint32_t lFrmCnt = 0;

while((lpFrame = lIsp.GetFrame()))
{
  lpFrame = OAL_MemoryReturnAddressPhysical(
                          lpFrame,
                          ACCESS_PHY + 1); // get virtual address
  lDcuOutput.PutFrame(lpFrame, false);

  lFrmCnt++;
}
```

To terminate the ISP graph execution the `FrameInputISP::Stop()` method has to be called.

```
lIsp.Stop();.
```

4 High Level Design

4.1 System Decomposition

The `FrameInputISP` object is part of the `frame_io` library and creates a temporary solution to access the ISP configuration and preprocessed data from the user application. It is derived from the `FrameInputBase` object which defines the least supported API.

4.2 File Structure

`FrameInputISP` object code is located in VSDK under s3234_sdk/libs/io/frame_io folder. Internally it has the following structure:

- build-* – build folders for supported platforms (standalone and Linux)
  - Makefile
- include
  - frame_input_isp.h – declaration of the object and its API,
- src
  - frame_input_isp.cpp – definition of the object and its API,
- BUILD.mk – defines build details

4.3 Module Usage

< This section contains module usage restrictions.>