Document Type:

Application Note - VisionIP DOF HIL/SIL

Project Name:

Vision IP Dense Optical Flow

Responsible Department	Global ADAS Solution Group
Approved by	
Checked by	
Prepared by	A. Schulz

Document No. D017381
Date Published *n/a*

www.renesas.com

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose and Scope

This application note serves as a quick guide into Dense Optical Flow (DOF) using Vision IP drivers. Background information of DOF is detailed in chapter 2 and example code is described in chapter 3.

The example code also demonstrates the compatibility between the two major XIL frameworks,

- SIL – simulator in the loop
- HIL – hardware in the loop

Apart from a few UDFs mostly for file I/O and memory management, no other dependencies regarding which framework is used exists in the code. The interaction with the configuration library and the system driver is identical for both frameworks.

1.2 Prerequisites

Hardware:

- V4H Starter Kit development board or similar

Software:

- Linux OS / BSP
- later-Car SDK1 for Windows and Linux

1.3 Terms, Abbreviations

Term / Abbreviation	
BSP	board support package
DOF	dense optical flow
HIL	hardware in the loop
MIL	model in the loop
PIL	processor in the loop
ROI	region of interest
SIL	simulator in the loop
UDF	user defined function
XIL	umbrella for HIL, MIL, PIL and SIL

2 Dense Optical Flow

2.1 Overview

The Dense Optical Flow is a process to extract pixel movement information between two related images. The output of this process are at max, two flow fields indicating where each pixel in the 1st image can be found in the 2nd one and vice versa.

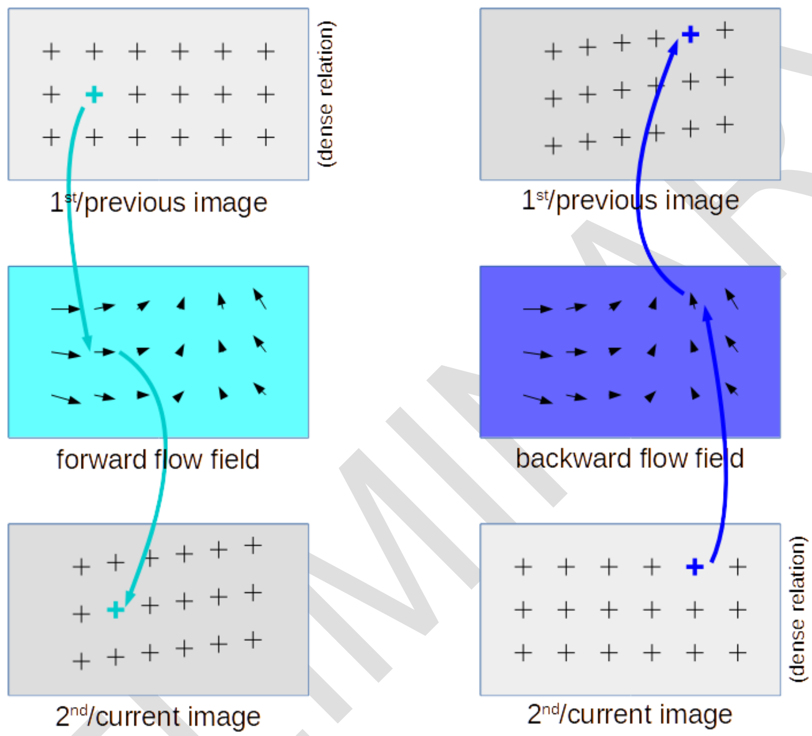

Figure 2-1 relation of input images to flow field

2.2 Applications

2.2.1 Movement of Objects

Using one and the same input source (e.g. a single camera setup), the result of DOF of two consecutive images interprets to movements of objects over time.

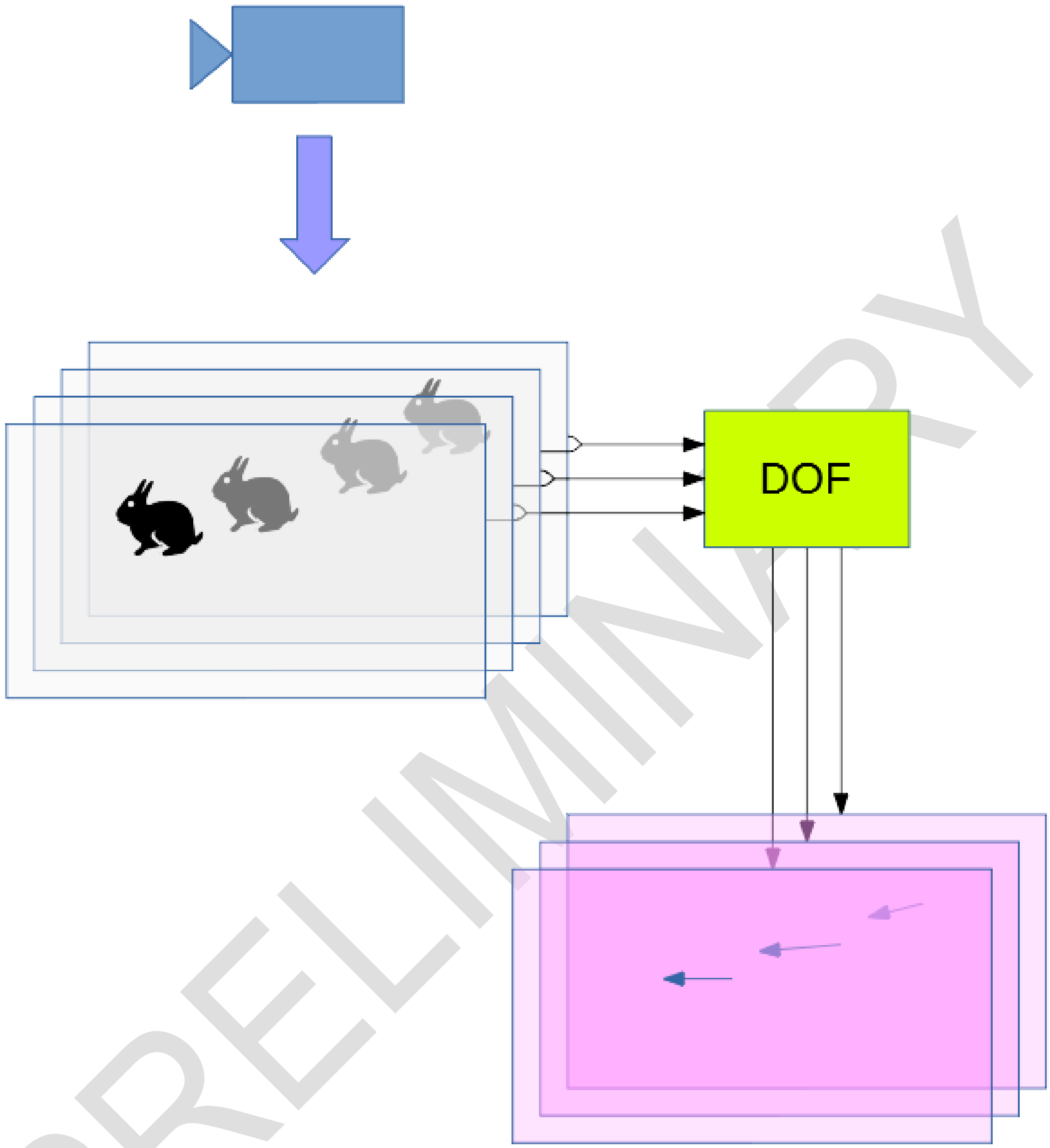

Figure 2-2 single camera setup – object motion

Figure 2-3 single camera setup – real world scenario (input image (1/2))

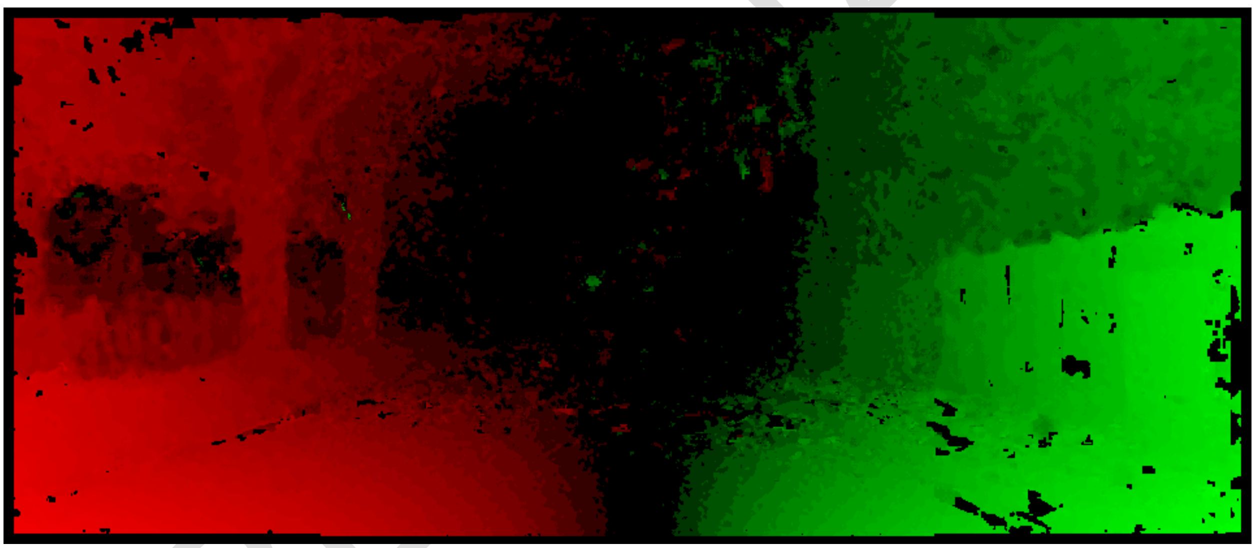

Figure 2-4 single camera setup – real world scenario (flow field (horizontal))

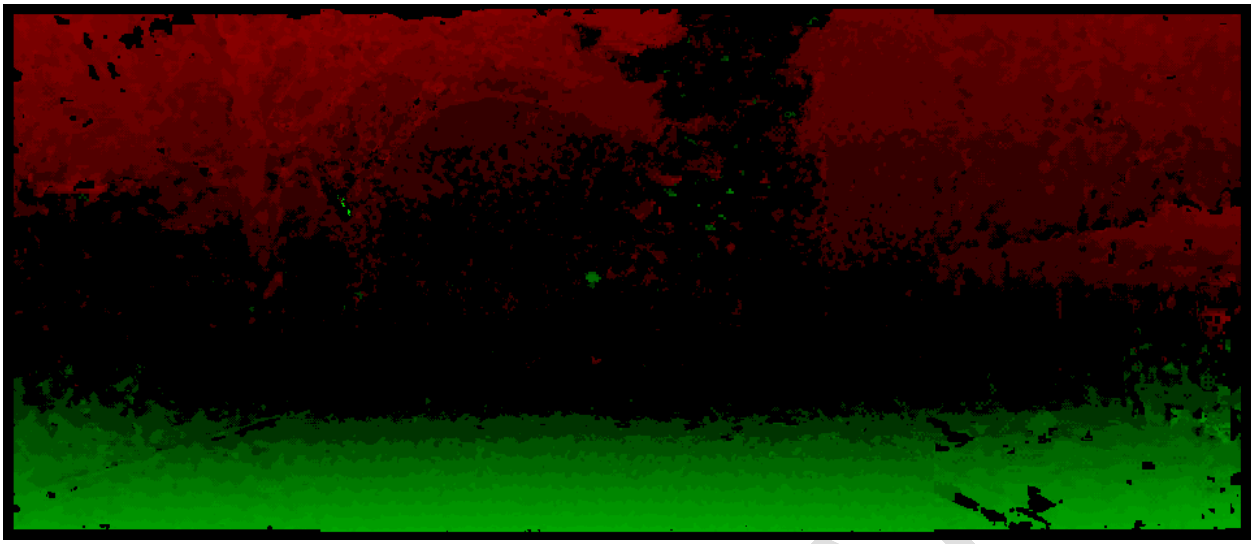

Figure 2-5 single camera setup – real world scenario (flow field (vertical))

2.2.2 Distance of Objects

With a synchronized dual camera setup, processing DOF on their images, taken at the same time, produces a flow field which is related to the distance of objects.

Hereby, for horizontally aligned cameras, only the horizontal flow field vector component carries the desired data. The vertical component, in best case, is zero for all vectors.

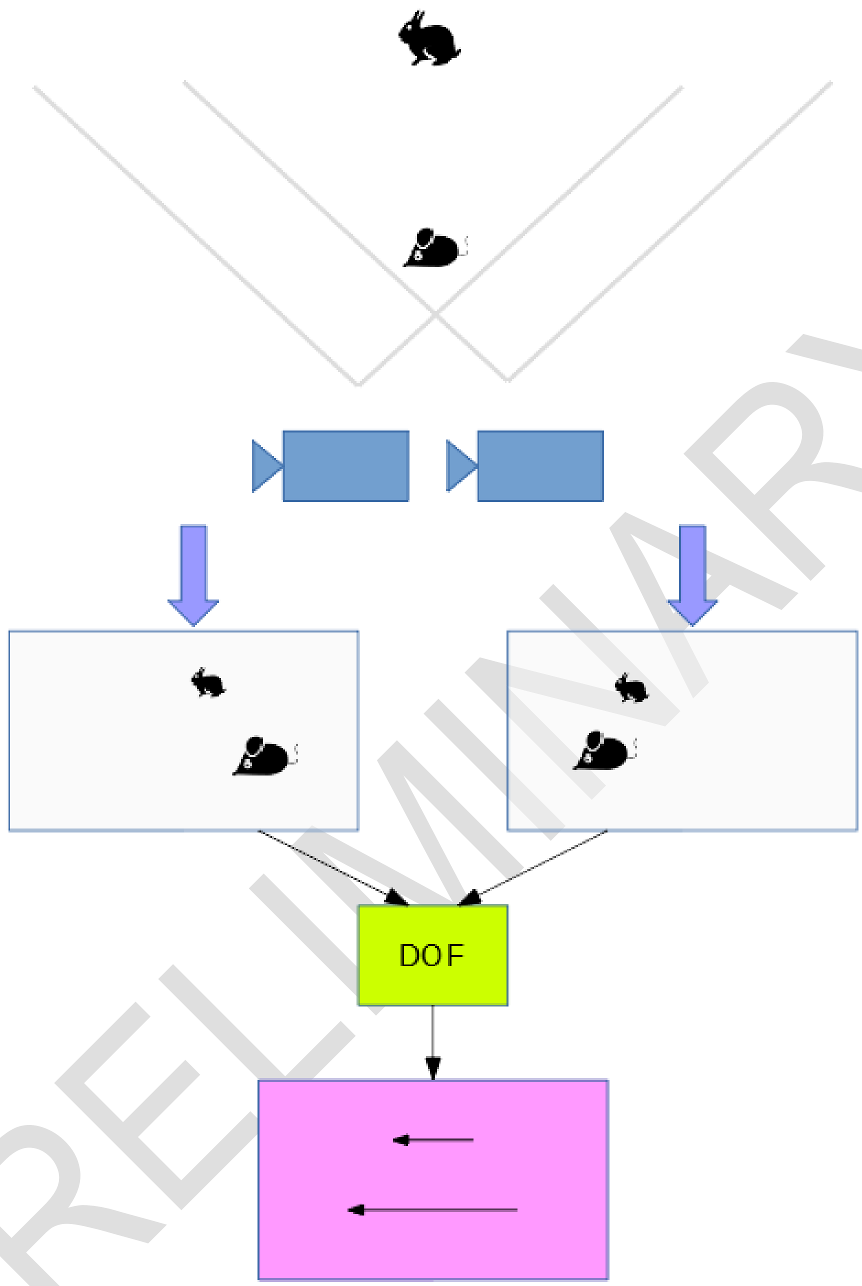
DOF

Figure 2-6 dual camera setup – real world scenario (input image (1/2))

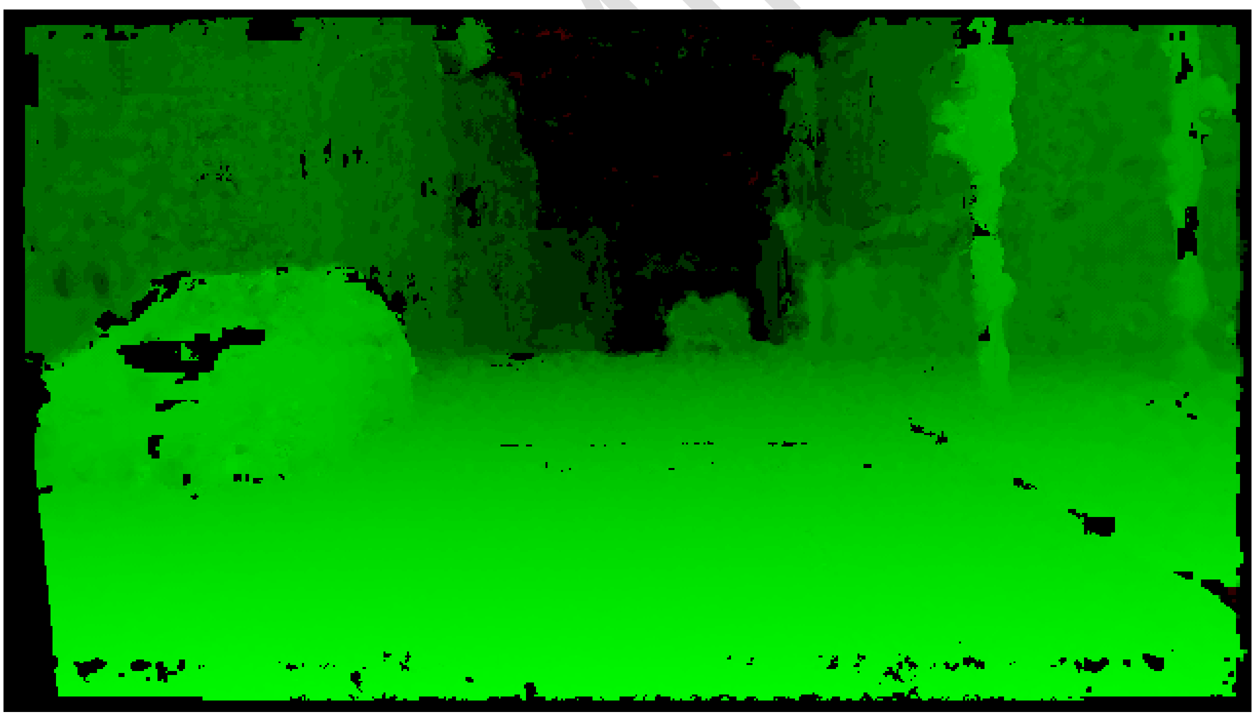

Figure 2-7 dual camera setup – real world scenario (flow field (horizontal))

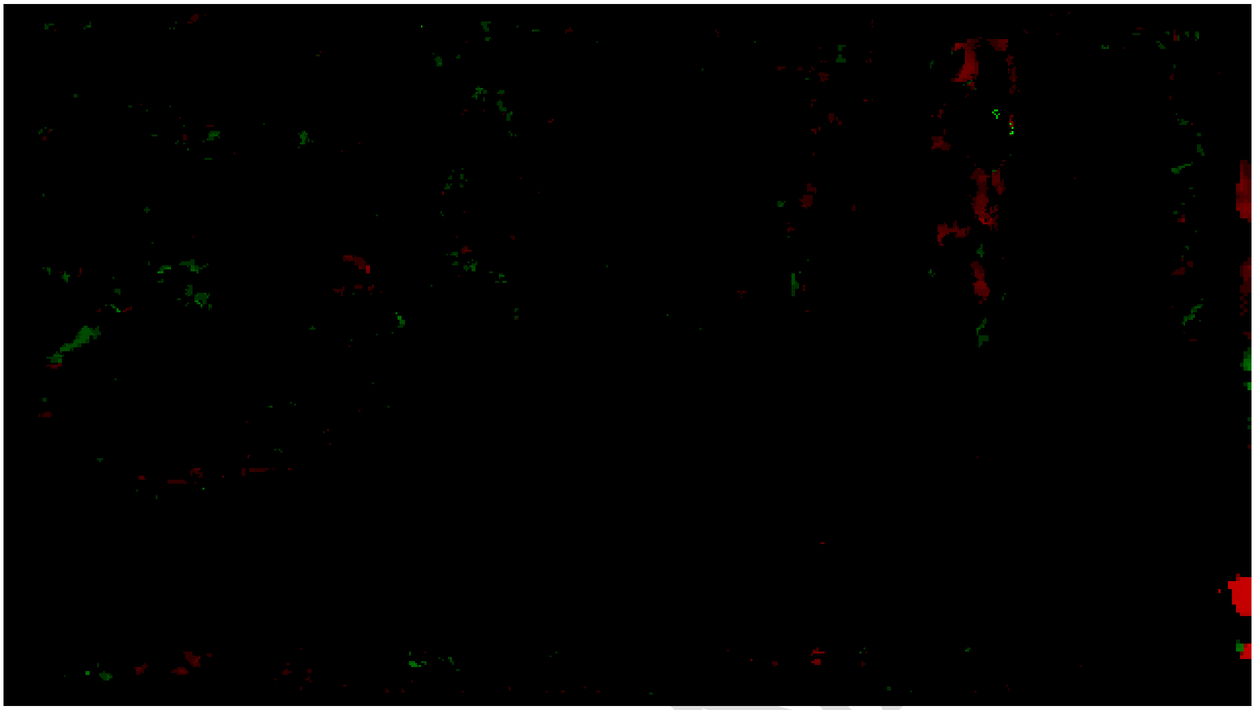

Figure 2-8 dual camera setup – real world scenario (flow field (vertical))

2.3 Process

Described below is the processing for the forward flow. Regarding backward flow, the process is the same only with 1st and 2nd image exchanged.
For **each** pixel in the 1st image, the DOF process uses a search window as an area to search for the matching pixel in the 2nd image.

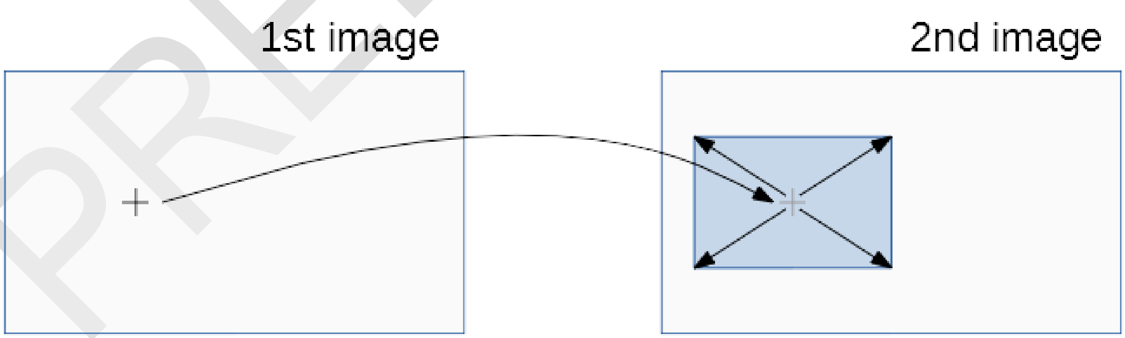

Figure 2-9 search window

The difference in the coordinates of the matching pixel between the 1st and the 2nd image yields the flow vector result.

Calculated for all pixels in the 1st image, a 2-dimensional flow field is created for the entire input image.

As the size of the search window is directly related to the processing effort/time, the DOF process is intended to be executed on up to 3 different resolutions, usually with 1:1, 1:2 and 1:4 scaling.

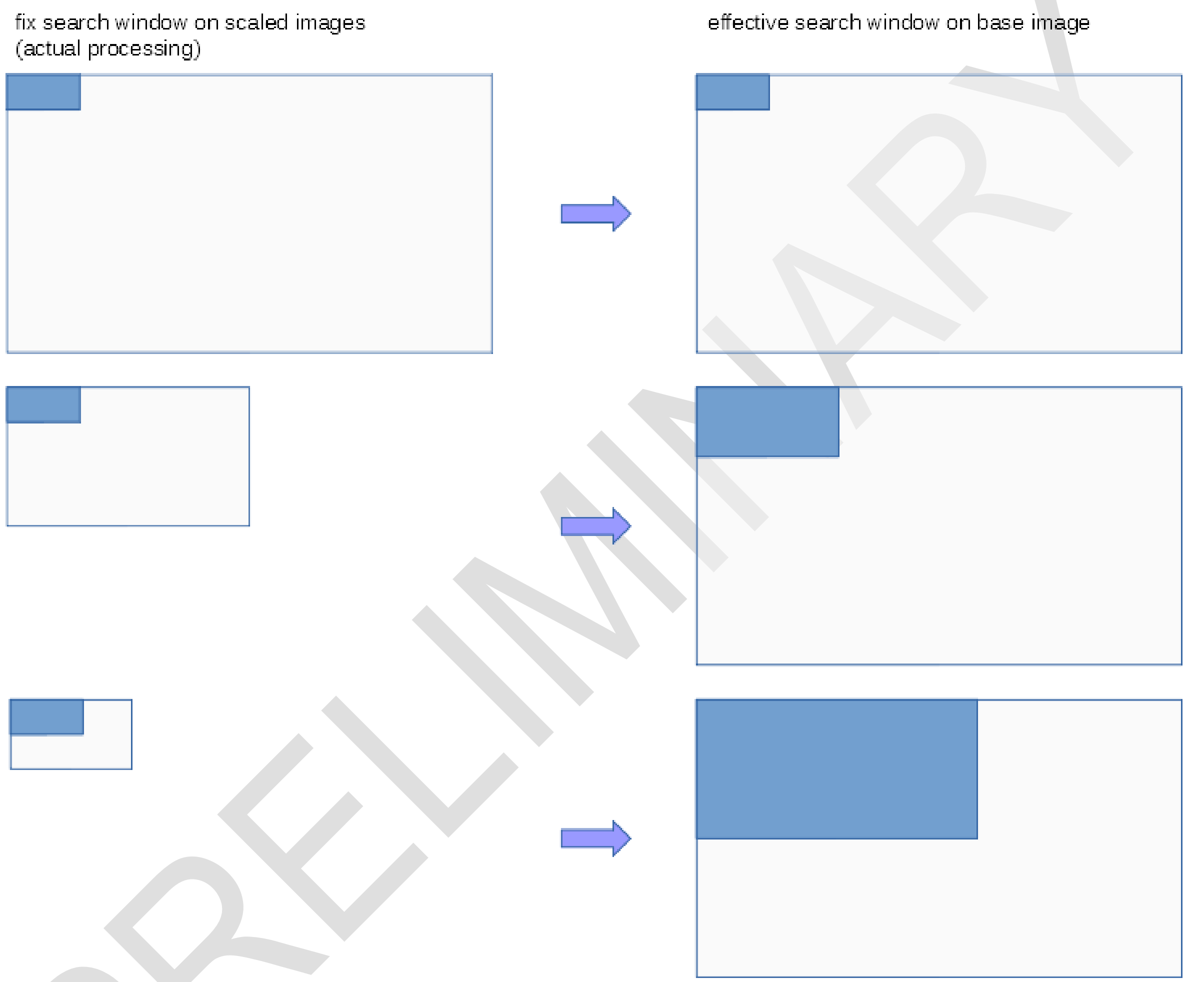

Figure 2-10 search window at different resolutions

Using 3 different resolutions hereby reduces the computational effort:

single execution with required search window:

- resolution factor: 1
- search window factor: 1

⇨ overall factor: $1 * 1 = 1$

3 executions with scaled resolutions and scaled search windows

- 1^{st} execution
  - resolution factor: $(1/4)^2$
  - search window factor: $(1/4)^2$
- 2^{nd} execution
  - resolution factor: $(1/2)^2$
  - search window factor: $(1/4)^2$
- 3^{rd} execution
  - resolution factor: 1
  - search window factor: $(1/4)^2$

⇨ overall factor: $\left(\frac{1}{4}\right)^2 * \left(\frac{1}{4}\right)^2 + \left(\frac{1}{2}\right)^2 * \left(\frac{1}{4}\right)^2 + 1 * \left(\frac{1}{4}\right)^2 \sim 0.082$

In order to merge the result of the 3 processing's, the DOF hardware IP contains a fusion stage which merges the result of a previous processing on the fly.

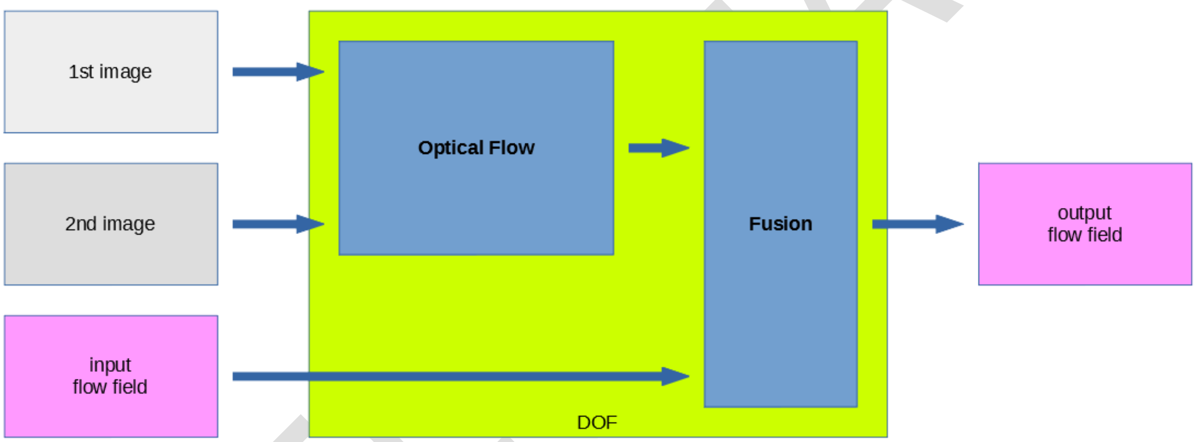

Figure 2-11 optical flow and fusion

Since input and output flow field are usually at different resolutions, the fusion stage is equipped with on-the-fly up & down scaling blocks, usable as necessary.

3 Application

3.1 Content

t.b.d.

3.2 Integration

t.b.d.

3.3 Usage

t.b.d.

3.4 Operation

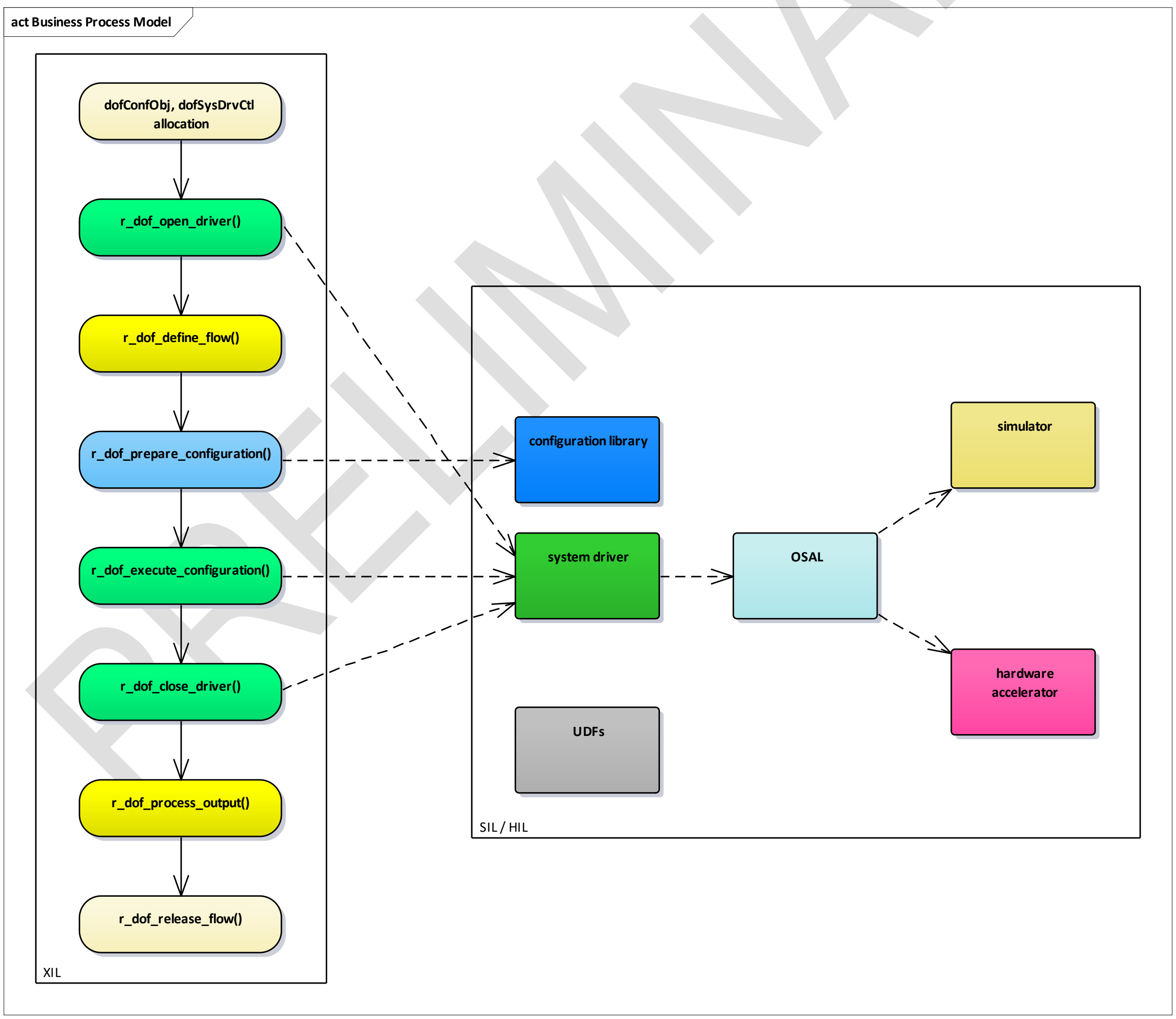

Figure 3-1 DOF test

Both, SIL and HIL implementation of this application note start with the main() function which calls the actual DOF test.
The DOF test, executes various functional blocks. In addition indicated, the majority of the test is assigned to the XIL layer and therefore identical for both SIL and HIL variant of this test. The differences are limited to the caller of this test (i.e. the main() function) and the implementation of the OSAL and layers below.

3.4.1 Flow definition

Prior to flow definition, the application defines the following few key points:

- preferred L1 input image width
- preferred L1 input image height
- downscaling of L1 to L2
- backward flow, forward flow or both in parallel

With these parameters, the flow definition hereby determines all remaining parameters automatically.
i.e. for all levels L3, L2 and L1 and for each level-based input image, input flow (L2 & L1 only) and output flow:

- resolution of frame
- resolution of ROI
- position of ROI within frame

Based on the option for downscaling L1 to L2, the two scenarios exist:

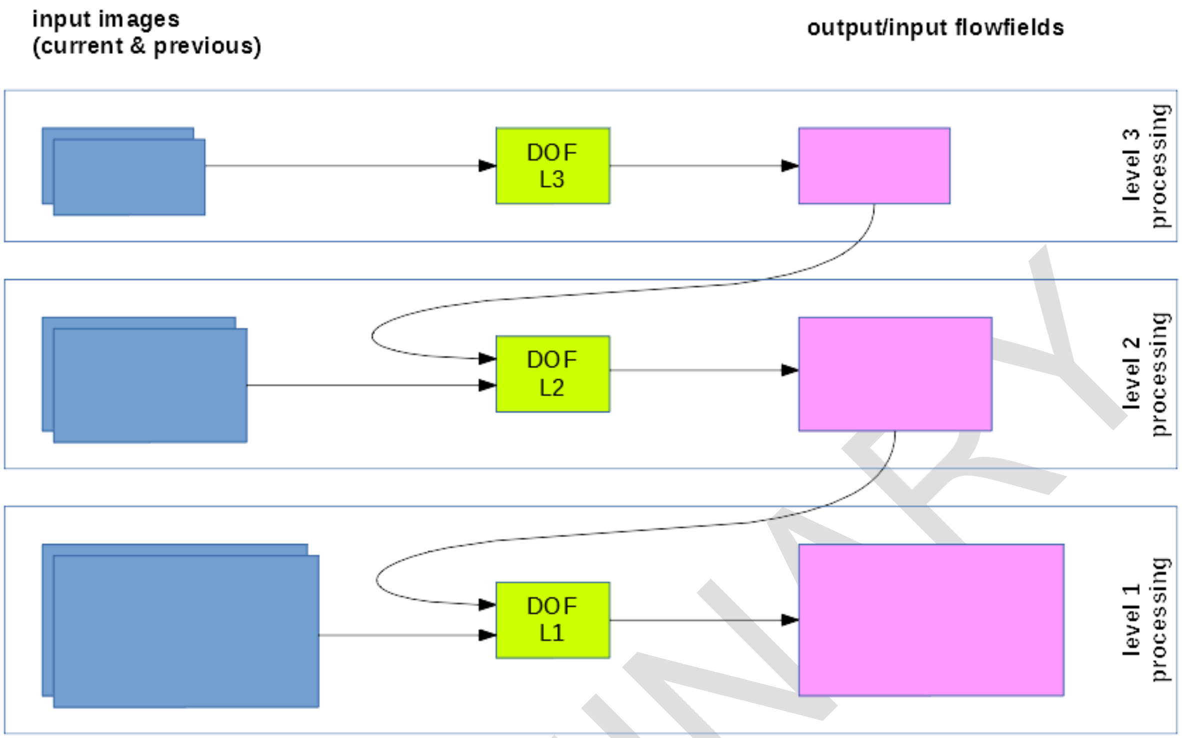

Figure 3-2 separate buffers, no L1 to L2 downscaling

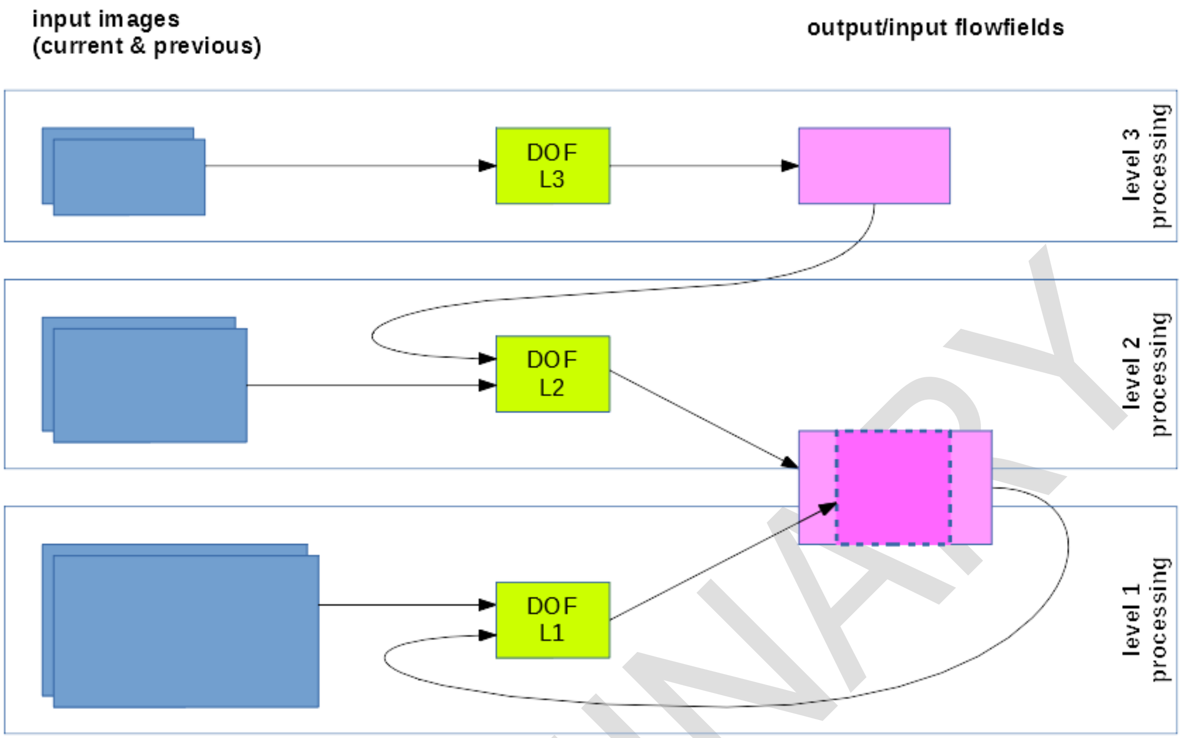

Figure 3-3 no dedicated L1 flow field buffer, with L1 to L2 downscaling

Along with the considerations of the hardware accelerator limitations (e.g. alignments, min/max resolutions) and dependencies among the levels, the target for solving is to achieve the optimal case for the complete flow with minimum buffer sizes without dropping any valid input image pixels.

After successful solving, the flow definition prints the buffer requirements.

3.4.2 Configuration preparation

The preparation of the configuration object is separated in three major blocks:

- initialization of the configuration object
    - setup of default configuration values for all levels L3, L2 & L1
      note: this also predefines configuration values not explicitly set in later steps
- conditioning of configuration object
    - apply area information, calculated by the flow definition in 3.4.1
    - allocation and assignment of output flow fields
    - input buffer setup
        - allocation, loading (file I/O) and assignment of input images
        - assignment of input flow fields by reusing output flow fields
- finalizing of the configuration object
    - execution of the configuration libraries finalize function for post checking and ready-for-execution marking

As the result of this configuration preparation, a configuration object is available, ready for execution by simulator / hardware accelerator.

3.4.3 Configuration execution

With the prepared configuration object, the simulator / hardware accelerator is invoked for callback-based execution.
Upon DOF processing end, the assigned callback function is executed, testing for the status / result of the execution and notifying the main thread of the application to continue processing.

3.4.4 Post processing

After execution has finished successfully, export files are created by the post processing functions.

3.4.4.1 Raw data

For each level, a dump of the entire flow field (at frame resolution, including the ROI) is exported as-is without any additional file header or footer.
Therefore, the content of this export is an area of flow field vectors, each 32 bit with the bit utilization as following:

Table 3-1 Flow field vector output

bit	description
31:22	horizontal motion component (data encoding, see FLE)
21:12	vertical motion component (data encoding, see FLE)
11:10	FLE – flow encoding specifies data encoding used for horizontal and vertical motion component: 0: sQ5.4 (range: -32…31.9375, accuracy: 1/32) 1: sQ6.3 (range: -64…63.875, accuracy: 1/16) 2: sQ7.2 (range: -128…127.75, accuracy: 1/8) 3: sQ8.1 (range: -256…255.5, accuracy: 1/4)
9:3	reserved
2	PCD – pyramid confirmed: 0: flow vector at Ln conflicts with or has not result at Ln+1 1: flow vector at Ln corresponds to result at Ln+1 (best case)
1:0	PLO – pyramid level origin: 0: no level, flow vector is invalid 1: L1 (original resolution) (best case) 2: L2 (1/2 resolution) 3: L3 (1/4 resolution)

3.4.4.2 Colorized single component flow fields

For visualization, a horizontal and a vertical image file is created, showing each component's directional magnitude by color brightness.

Hereby red and green are used to indicate negative resp. positive component values with the brightness referring to the magnitude of the component.

The overall brightness of the images is logarithmically upscaled to a maximum value.
The maximum value used for this upscaling is chosen from the maximum of both horizontal and vertical component values of all vectors for a single level to keep both resulting images (horizontal and vertical component image) visually comparable regarding their brightness.

3.4.4.3 Level maps

The level maps for each level provide a visual indication of the validity and the origin of the flow vector. This map basically indicates the selections the fusion stage of the DOF has taken.

The color encoding is as following:

- black – flow vector is invalid, no result exists
- white – flow vector from L3
- light gray – flow vector from L2
- dark gray – flow vector from L1

3.4.4.4 Confirmation maps

Confirmation maps for L2 and L1 provide quality information of the flow fields based on the PCD attribute of each flow vector:

- black – flow vector is invalid (after fusion)
- yellow – flow vector at Ln is valid but not at Ln+1 or vector at Ln does not correspond to vector at Ln+1
- green – flow vector at Ln and Ln+1 are valid and correspond to each other

3.5 Re-Use / Extension

3.5.1 Additional configuration parameters

As noted in 3.4.2, not all possible parameters are set in the given example.
In order to change any of the remaining default values (e.g. search ranges), code adaptions targeting the "dofConfObj" can be made anywhere between calls for "r_initialize_configuration_object()" and "r_finalize_configuration_object()" using the configuration libraries function "R_DOF_ConfLibChangeParam()" as exercised in "r_apply_flow()".

3.5.2 Dedicated configuration

For a complete dedicated configuration, using own flow parameter, input and output buffers,

- r_dof_define_flow()
- r_dof_prepare_configuration()

can be completely replaced.

The minimum requirement here is to construct a valid configuration object "dofConfObj" (including the finalization via configuration library) before starting execution of it.

Optionally when setting up the "flowDescriptor" as well, describing buffer properties, the post processing function can be used as-is for visualization purposes.

3.5.3 Partially fixed configuration

In case all flow parameters are known (excluding buffer pointers), the configuration object can be serialized to a non-volatile memory and reloaded later for execution.

Assuming the post processing is not needed, it is only necessary to continue with the assignments of all input and output buffers, finalize the configuration object and start execution.
The complete flow definition, initialization of the configuration object as well as applying any buffer size information can be skipped.

3.5.4 Fully fixed configuration

In addition to 3.5.3, if also buffer locations are statically fixed (virtual & physical address are fix), using a serialized **finalized** configuration object allows execution of the simulator / hardware accelerator without any interacting with the configuration library.
This scenario is the fastest possible regarding configuration setup **and** execution.

3.5.5 Streamed ring buffer processing

The example code for this application note is a one-shot case.
A single configuration is created from scratch, executed while the necessary driver interactions are made.

For a use case involving multiple different executions, the process of creating configuration objects is different but can benefit from the various stages as shown next.

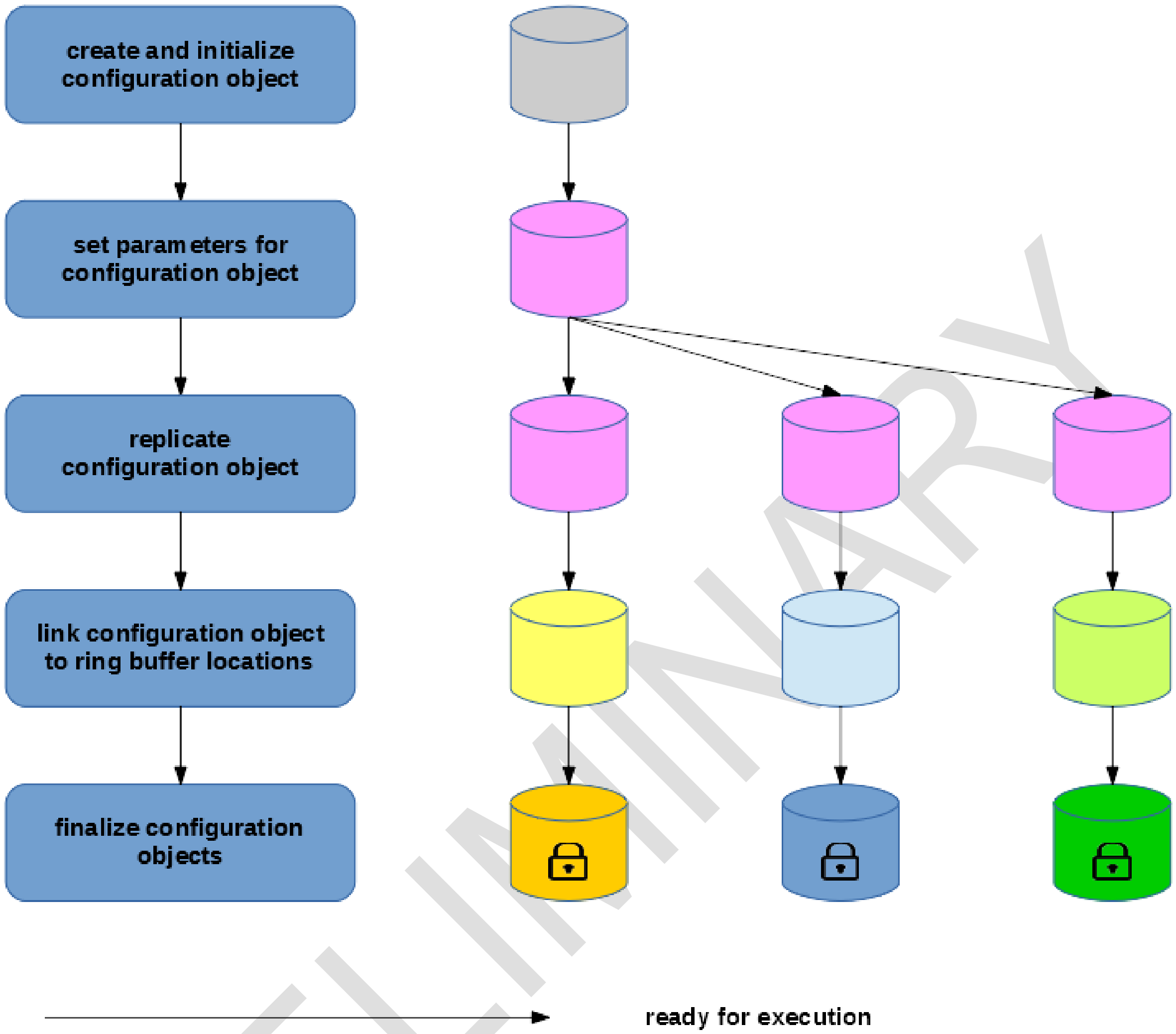

Figure 3-4 multiple similar configuration objects

For a partially fixed configuration, the configuration object can be replicated and afterwards the single instances can be assigned to different buffer locations (e.g. addresses within one and the same ring buffer). Beneficial hereby is to reduce the number of interactions with the configuration library and its overhead to process one and the same task multiple times.

In addition, having as many configuration objects as different buffer locations exist within a ring buffer allows to completely avoid reconfiguration during processing.

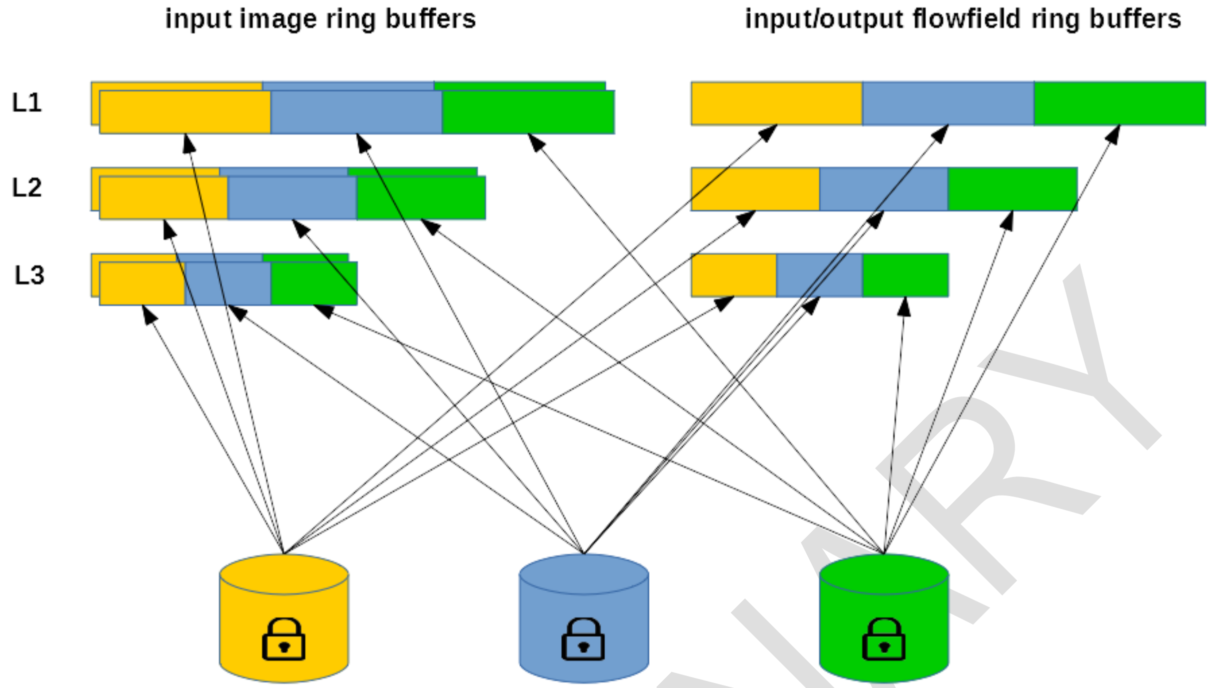

Figure 3-5 configuration objects assigned to ring buffer

Building blocks of this application note which might be usable for this scenario are:

- system driver handling
  - r_dof_open_driver()
  - r_dof_close_driver
- setup of base configuration object
  - r_dof_define_flow()
  - r_initialize_configuration_object()
  - r_apply_flow()
- completion of replicated configuration objects
  - r_prepare_output()
  - r_prepare_input()
  - r_finalize_configuration_object()

4 Open Issues

- Application note in preliminary state.
- functions and parameter types deviate from current implementation

5 Appendix

5.1 Configuration Details

5.1.1 Separation

The DOF driver supports at max 3 executions in a row. Usually, these executions refer to one flow and cover the level L3, L2 and L1, processing one input frame in pyramidal resolutions.
Since the driver and the IP keep the executions separated from each other, deviations from the default use case are possible.

e.g.

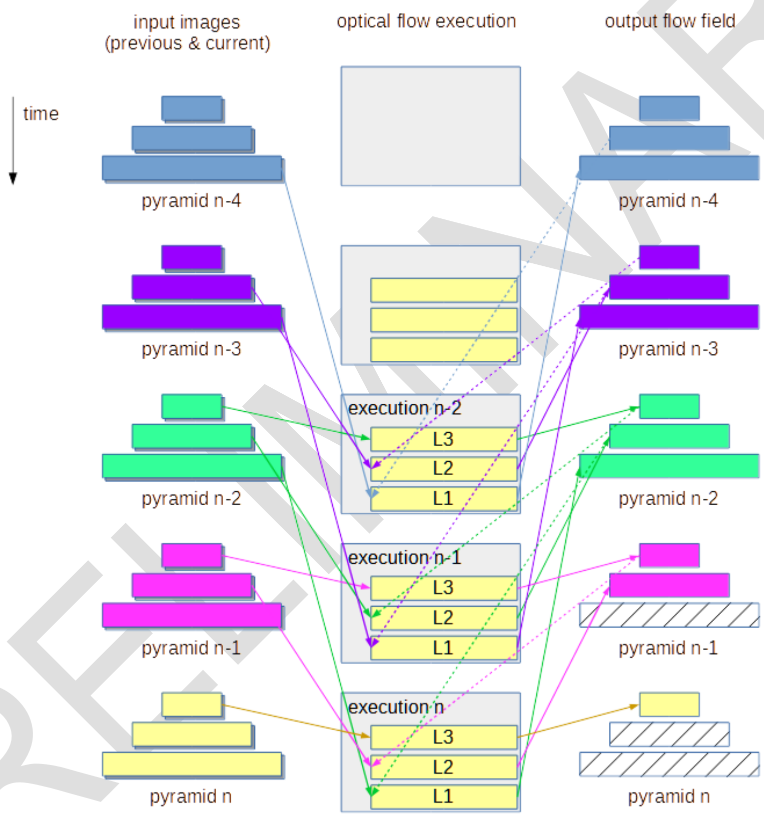

Figure 5-1 shifted pyramid processing approach

Though the latency of having the final result increases, this flow allows adapting the configuration parameters after each execution for a certain pyramid level separately.

To support the execution separation, all executions are configured independently from each other via dedicated execution level parameter for all configuration functions.

5.1.2 Scatter / Gather DMA usage

Input images, input flow field and output flow field are streamed via 2D scatter/gather DMA.
Providing the overall frame resolution as well as ROI configuration allows processing images/flow fields which extend DOF core hardware limits.

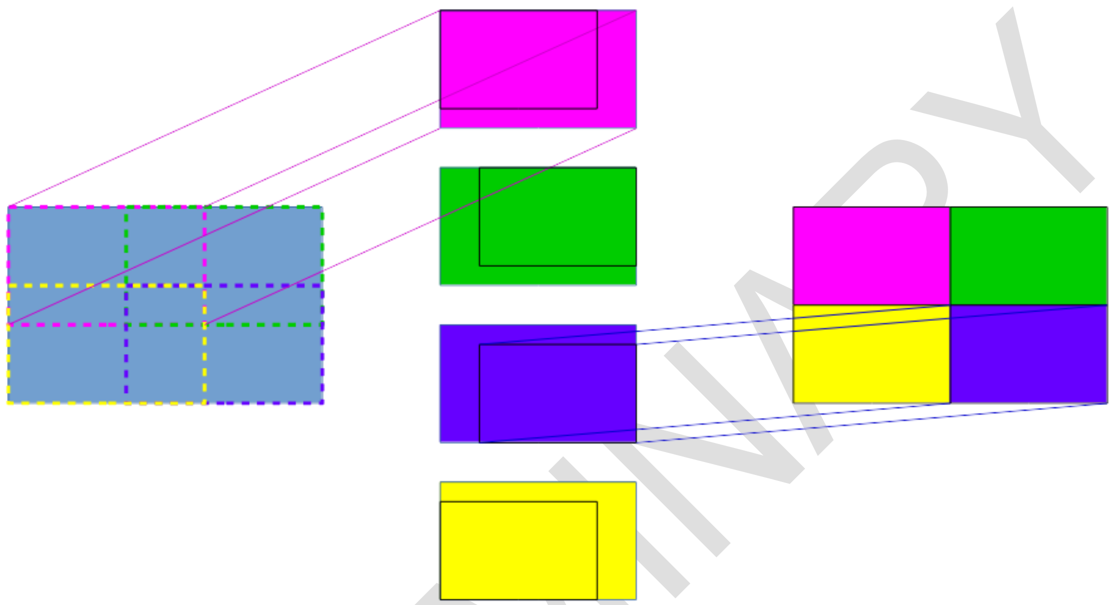

Figure 5-2 tile processing example

Note: For tile processing to be accurate, tiles need to overlap with their search ranges.

5.1.3 Flow field fusion

The configuration of the integrated fusion stage is automatically done by the driver.
Forward or backward fusion is active for a certain pyramid level as soon as an input flow field is provided.

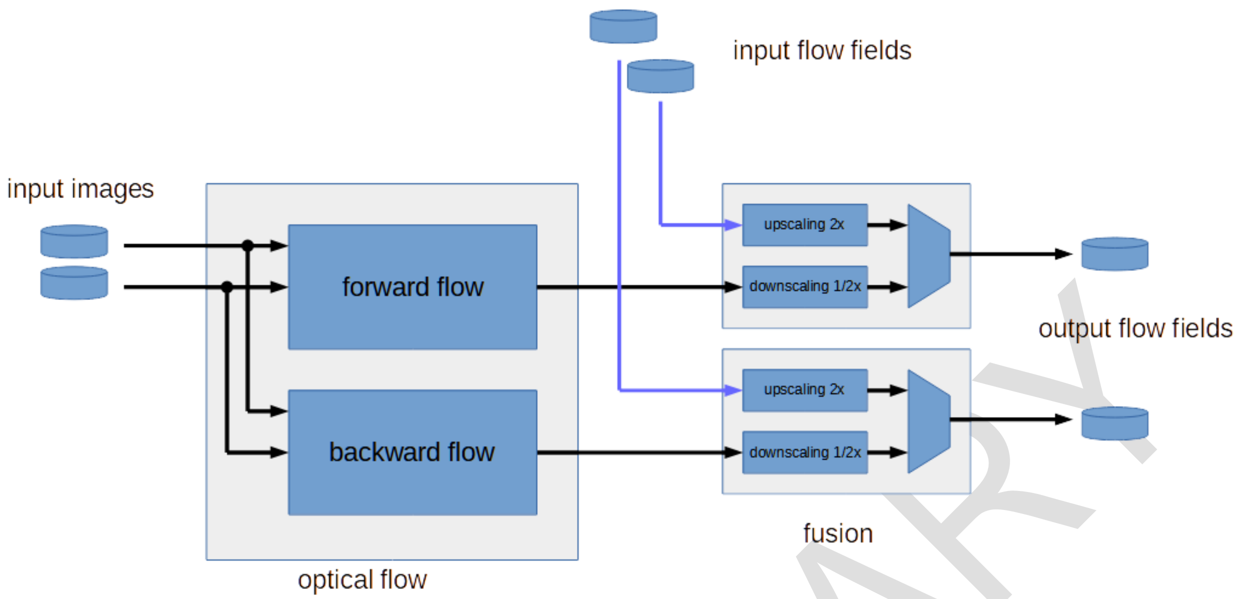

Figure 5-3 fusion stage

The configuration of the optional input flow field upscaling and intermediate flow field downscaling is derived from the resolutions of:

- input images and output flow field => intermediate flow field downscaling
- input flow field and output flow field => input flow field upscaling

5.1.4 Functions and Parameters

5.1.4.1 Input images

buffer setup; address & size:

necessity	mandatory	
related function	`R_DOF_ConfLibSetInputImages()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`uint16_t* p_curr_frame`	ptr to buffer of current image
	`uint16_t* p_prev_frame`	ptr to buffer of previous image
	`uint32_t frame_size`	byte size of each image buffer

buffer setup; resolution:

necessity	optional	
related function	`R_DOF_ConfLibChangeParam()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`e_dof_param_type_t type`	type of parameter
	`int32_t val`	value of parameter
related parameters types	`DOF_PAR_IN_CURR_FRAME_SIZE`	pixel count of buffer for current image

	`DOF_PAR_IN_PREV_FRAME_SIZE`	pixel count of buffer for previous image
	`DOF_PAR_IN_ROI_FRAME_WIDTH`	width of buffer for current image
	`DOF_PAR_IN_ROI_FRAME_HEIGHT`	height of buffer for current image

ROI setup:

necessity	optional	
related function	`R_DOF_ConfLibChangeParam()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`e_dof_param_type_t type`	type of parameter
	`int32_t val`	value of parameter
related parameters types	`DOF_PAR_IN_ROI_WIDTH`	width of ROI within buffer for current & previous image
	`DOF_PAR_IN_ROI_HEIGHT`	height of ROI within buffer for current & previous image
	`DOF_PAR_IN_ROI_HORI_OFF`	horizontal offset of ROI within buffer for current & previous image
	`DOF_PAR_IN_ROI_VERT_OFF`	vertical offset of ROI within buffer for current & previous image

5.1.4.2 Input flow field

buffer setup; address & size:

necessity	optional	
related function	`R_DOF_ConfLibSetInputFlowfields()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`uint32_t* p_fwd_flow_field`	ptr to buffer of input forward flow field
	`uint32_t* p_bwd_flow_field`	ptr to buffer of input backward flow field
	`uint32_t flow_size`	byte size of input forward and backward flow field buffers

buffer setup; resolution:

necessity	optional	
related function	`R_DOF_ConfLibChangeParam()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`e_dof_param_type_t type`	type of parameter
	`int32_t val`	value of parameter

related parameters types	`DOF_PAR_INFLW_FRAME_SIZE`	flow vector count of buffer for input forward and backward flow field
	`DOF_PAR_INFLW_ROI_FRAME_WIDTH`	width of buffers for input forward and backward flow field
	`DOF_PAR_INFLW_ROI_FRAME_HEIGHT`	height of buffers for input forward and backward flow field

ROI setup:

necessity	optional	
related function	`R_DOF_ConfLibChangeParam()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`e_dof_param_type_t type`	type of parameter
	`int32_t val`	value of parameter
related parameters types	`DOF_PAR_INFLW_ROI_WIDTH`	width of ROI within buffers for input forward and backward flow field
	`DOF_PAR_INFLW_ROI_HEIGHT`	height of ROI within buffers for input forward and backward flow field
	`DOF_PAR_INFLW_ROI_HORI_OFF`	horizontal offset of ROI within buffers for input forward and backward flow field
	`DOF_PAR_INFLW_ROI_VERT_OFF`	vertical offset of ROI within buffers for input forward and backward flow field

5.1.4.3 Output flow field

buffer setup; address & size:

necessity	mandatory	
related function	`R_DOF_ConfLibSetOutputFlowfields()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`uint32_t* p_fwd_flow_field`	ptr to buffer of output forward flow field
	`uint32_t* p_bwd_flow_field`	ptr to buffer of output backward flow field
	`uint32_t flow_size`	byte size of output forward and backward flow field buffers

buffer setup; resolution:

necessity	optional
related function	`R_DOF_ConfLibChangeParam()`

related arguments	`e_dof_pyr_level_t level`	pyramid level
	`e_dof_param_type_t type`	type of parameter
	`int32_t val`	value of parameter
related parameters types	`DOF_PAR_OUTFLW_ROI_FRAME_WIDTH`	width of buffers for output forward and backward flow field
	`DOF_PAR_OUTFLW_ROI_FRAME_HEIGHT`	height of buffers for output forward and backward flow field

ROI setup:

necessity	optional	
related function	`R_DOF_ConfLibChangeParam()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`e_dof_param_type_t type`	type of parameter
	`int32_t val`	value of parameter
related parameters types	`DOF_PAR_OUTFLW_FRAME_SIZE`	flow vector count of buffer for input forward and backward flow field
	`DOF_PAR_OUTFLW_ROI_WIDTH`	width of ROI within buffers for output forward and backward flow field
	`DOF_PAR_OUTFLW_ROI_HEIGHT`	height of ROI within buffers for output forward and backward flow field
	`DOF_PAR_OUTFLW_ROI_HORI_OFF`	horizontal offset of ROI within buffers for output forward and backward flow field
	`DOF_PAR_OUTFLW_ROI_VERT_OFF`	vertical offset of ROI within buffers for output forward and backward flow field

5.1.4.4 Search ranges

necessity	optional	
related function	`R_DOF_ConfLibChangeParam()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`e_dof_param_type_t type`	type of parameter
	`int32_t val`	value of parameter
related parameters types	`DOF_PAR_SEARCH_RANGE_LEFT`	search window left border distance relative to flow vector coordinates

	`DOF_PAR_SEARCH_RANGE_RIGHT`	search window right border distance relative to flow vector coordinates
	`DOF_PAR_SEARCH_RANGE_UP`	search window top border distance relative to flow vector coordinates
	`DOF_PAR_SEARCH_RANGE_DOWN`	search window bottom border distance relative to flow vector coordinates
	`DOF_PAR_VERTICAL_SHIFT`	vertical (down) shift of overall search window

5.1.4.5 Image pre filtering

necessity	optional	
related function	`R_DOF_ConfLibSetSmcCoeff()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`uint8_t *p_ft_c`	ptr to 3x3 matrix with filter kernel for current input image
	`uint8_t *p_ft_p`	ptr to 3x3 matrix with filter kernel for previous input image

5.1.4.6 Flow vector shifting

necessity	optional	
related function	`R_DOF_ConfLibSetBitShiftEn()`	
related arguments	`e_dof_pyr_level_t level`	pyramid level
	`uint32_t value`	digits to left shift each flow vector before fusion stage

Note:
Using a large search range and a large flow vector shifting may lead to overflowing vectors which cannot be encoded in the output anymore.
In this case, the IP will trigger an error interrupt to indicate this issue to the application.

6 References

There are no sources in the current document.

7 Revision History

Table 7-1 Revision History

Version	Details	Chapter	Release Date	Prepared by	Approved by
1.0	initial document release	all	06.12.2019	A. Schulz	
(2.0)	update for OMM 1.1	t.b.d.	n/a	A. Schulz	