DVS Digital Video Systems AG

DVS SDK - DVS Render API Extension

Supplement Reference Guide

DVS Digital Video Systems AG
Version 1.2d supplementing the "DVS SDK" Reference Guide 4.0.x

Table of Contents

DVS SDK Main Page

Introduction

This document describes the DVS Render API Extension. It is a set of defines, functions and structures that extend the DVS SDK (DVS Software Development Kit) enabling you to perform memory to memory video processing operations in hardware.

Target Group

To use this guide and the DVS SDK you should have experience in software development and knowledge in the field of digital video/audio in general, including knowledge about the handling and the internal structure of a digital video system.

Furthermore, you should know how to work with the DVS video device at hand as well as how to handle its driver.

Conventions Used in this Reference Guide

The following typographical conventions will be used in this documentation:

- Texts preceded by this symbol are parts of a list, first level as well as subordinated levels.

italic	Functions, parameter names (variables) or structures (structs).
`typewriter`	Defines, values, code examples, or commands (e.g. in your code).
`typewriter italic`	Programs, directories or directory structures, or files.

`<xxx>` is a place holder. If it is used, for example, with an option call or flag, it indicates a group of at least two of these calls/flags.

`<a>..<b>` indicates a value range from value `<a>` to value `<b>`.

General Information

This section contains some general information about the DVS Render API Extension and this reference guide.

Note:

This document is a supplement to the "DVS Software Development Kit" reference guide. For any further information not described here please refer to this reference guide as well as all other guides and manuals delivered with your DVS video board product.

Most structures and parameter defines are documented in the source code of the DVS SDK directly. For further information about a structure or parameter define not described in this reference guide please refer to its comments in the respective header file of the DVS SDK.

Supported DVS Video Board Products

The following DVS video board products are supported by the DVS Render API Extension:

DVS Video Board Product	**Serial Number** (first two digits)
Atomix	27
Atomix LT*	5 BNCs: 41 4 BNCs and D-Sub: 42

* The LT versions are in most respects identical to their respective counterpart without 'LT'. Therefore, in this reference guide they will be subsumed under the name of their counterpart, meaning e.g. whenever 'Atomix' is mentioned 'Atomix LT' is meant as well.

What's New in this Supplement Reference Guide

The following details the major additions and changes that were made to this supplement reference guide in its latest revisions:

New in Version 1.2:

- Added sharpness to the structure *sv_render_scaler*.
- Implemented upscaling via the function *sv_render_push_scaler()*.
- Revised the limitation description of the function *sv_render_push_scaler()*.

New in Version 1.1:

(This version was only available as a draft.)

- Added the define `SV_RENDER_OPTION_SAFERENDER`.
- Added the functions *sv_render_option_set()* and *sv_render_option_get()*.
- Improved behavior of the function *sv_render_open()* to disable the outputs of the DVS video board if no output FIFO is used.
- Improved function *sv_render_free()* when using the Render API together with the FIFO API (see also chapter API – FIFO API – Usage with the Render API).

New in Version 1.0:

First supplement reference guide for the DVS Render API Extension.

Hardware Render Pipelines

This section describes the hardware render pipelines of the DVS video boards supported by the Render API (see Supported DVS Video Board Products). It details for each board separately the sequence of possible processing steps as they may be carried out by the hardware.

When a render operation is initiated with the function *sv_render_issue()*, the source buffer will be read from the memory of the DVS video board. Afterwards the rendering is performed in hardware, starting with processing module 1 and ending with module *n*. Modules whose corresponding settings (see chapter API – Render API – Stack Settings Functions) were not specified on the render stack are skipped. Once the rendering is finished, the result will be written to the target buffer.

Render Pipeline of Atomix

This section lists all processing modules of the render pipeline of Atomix:

Module	Name	Remark
1	JPEG2000	Available for Atomix JPEG2000 only
2	Matrix	3 * 3 plus 1 for alpha
3	1D LUT	12 bit
4	Scaler	
5	3D LUT	16 bit
6	Matrix	3 * 3 plus 1 for alpha

DVS SDK Module Index

DVS SDK Modules

Here is a list of all modules:

DVS SDK Data Structure Index

DVS SDK Data Structures

Here are the data structures with brief descriptions:

DVS SDK Module Documentation

API – Render API – Basics

Detailed Description

The Render API is a video processing API for video board integration customers. It allows you to perform memory to memory video processing operations, thereby converting and modifying images in hardware (hardware rendering).

This chapter provides first some general information about the Render API and its usage. This will be followed by descriptions of the basic functions of the Render API, for instance, functions to open and close a render handle.

Operation of the Render API

The Render API operates stack based, meaning specific elements can be pushed on a stack for processing. The elements that can be pushed on the stack are 'images', 'operators' and 'settings' (described in their respective chapters of this reference guide).

To achieve maximum processing speed you should parallelize the render requests via multiple render contexts (stacks), because the Render API uses an internal hardware pipelining. Render contexts can be set up with the function *sv_render_begin()*.

Currently we recommend to use between six and eight parallel threads (render contexts) to receive maximum performance.

Default Render API Calling Sequence

When starting your application, you require a direct handle to the DVS video board from the function *sv_open()* or *sv_openex()* (i.e. an *sv_handle* pointer). Once the handle is valid, you can open the Render API with the function *sv_render_open()*.

```
sv_handle * pSV = 0;
sv_render_handle * pRender = 0;

// Open the sv_handle pointer
pSV = sv_open("");
if(!pSV) {
  ... // Appropriate error handling
}

// Open the sv_render_handle pointer
res = sv_render_open(pSV, &pRender);
if(res != SV_OK) {
  ...
}
```

With the necessary global handles available you are now able to create a specific render context containing its own render stack. To create an *sv_render_context* handle use the function *sv_render_begin()*. You can create and work on more than one *sv_render_context* handle in parallel.

```
sv_render_context * pRenderContext = 0;

// Create the specific render context
res = sv_render_begin(pSV, pRender, &pRenderContext);
if(res != SV_OK) {
  ... // Appropriate error handling
}
```

After this the source buffer should be transferred via a DMA transfer (i.e. via the functions *sv_render_dma()* or *sv_render_dmaex()*) to the DVS video board for processing. However, prior to this you have to allocate virtual space on the DVS video board with the function *sv_render_malloc()*.

```
unsigned char   * pSourceData;
unsigned int      sourceDataSize;
sv_render_image * pSourceImage = 0;
sv_render_image * pTargetImage = 0;

// Allocate space for the source buffer on the DVS video board RAM
res = sv_render_malloc(pSV, pRender, &pSourceImage, ...);
if(res != SV_OK) {
  ... // Appropriate error handling
}

// Transfer the source buffer from the system RAM to the DVS video board RAM
res = sv_render_dma(pSV, pRender, true, pSourceImage, pSourceData, ...);
if(res != SV_OK) {
  ...
}

// Allocate space for the target buffer on the DVS video board RAM
// Needed to store the rendered image
res = sv_render_malloc(pSV, pRender, &pTargetImage, ...);
if(res != SV_OK) {
  ...
}
```

When all initialization functions are performed, you can push elements on the render stack. The render operation can be executed with the function *sv_render_issue()*. Afterwards you can transfer the result with a DMA transfer back to the system memory.

```
unsigned char * pTargetData;
unsigned int    targetDataSize;

// At this point you will call functions to push settings, images and operators
// on the render stack. This will be described in the next section.
...

// Start the render operation for real
res = sv_render_issue(pSV, pRender, pRenderContext, ...);
if(res != SV_OK) {
  ... // Appropriate error handling
}

// Transfer the target buffer back to the system memory
res = sv_render_dma(pSV, pRender, false, pTargetImage, pTargetData, ...);
if(res != SV_OK) {
  ...
}
```

If you need to render further frames you can reset the render context with the function *sv_render_reuse()*, or you can call *sv_render_end()* and create a new context with the function *sv_render_begin()*.

After the operations are finished you have to free the allocated buffers on the DVS video device with the function *sv_render_free()* and close all other opened handles with their respective counterpart functions.

Example Stack Calling Sequence

The following shows in an example how to create a stack. You may create the stack in the sequence as shown below. However, it will be processed in reverse order (from bottom to top in our example, Polish notation).

```
Stack (before sv_render_issue())
-------------------------------
1. Image0   (buffer)   - sv_render_push_image()
2. Setting  (matrix)   - sv_render_push_matrix()
3. Setting  (1D LUT)   - sv_render_push_1dlut()
4. Setting  (JPEG2000) - sv_render_push_jpeg2000_decode()
5. Operator (render)   - sv_render_push_render()
```

You can find detailed information about the image, settings and operator functions in the corresponding chapters of this reference guide.

Defines

- #define SV_RENDER_OPTION_SAFERENDER

Functions

- int sv_render_begin (sv_handle *sv, sv_render_handle *prender, sv_render_context **ppcontext)
- int sv_render_close (sv_handle *sv, sv_render_handle *prender)
- int sv_render_end (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext)
- int sv_render_issue (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, sv_overlapped *poverlapped)
- int sv_render_open (sv_handle *sv, sv_render_handle **pprender)
- int sv_render_option_get (sv_handle *sv, sv_render_handle *prender, int option, int *pvalue)
- int sv_render_option_set (sv_handle *sv, sv_render_handle *prender, int option, int value)
- int sv_render_ready (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, int timeout, sv_overlapped *poverlapped)
- int sv_render_reuse (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext)

Define Documentation

#define SV_RENDER_OPTION_SAFERENDER

This define returns a bit mask of activated error and safety checks that will be applied during a render process. The error and safety checks are activated by default. By disabling the checks you can increase the performance, for example, for previews on low performance systems. Then, however, some of the rendered frames may be corrupt.

Function Documentation

int sv_render_begin (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context ** *ppcontext*)

This function sets up a render context containing a processing stack of its own. You have to set up at least one render context after calling the function *sv_render_open()* before starting the processing. Its counterpart function is the function *sv_render_end()*.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

ppcontext – Returns a handle to the render context.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

Note:

This function is thread-safe and can be called from different thread contexts at the same time to perform parallel render operations.

int sv_render_close (sv_handle * *sv*, sv_render_handle * *prender*)

This function closes the global Render API handle. After this call the *sv_render_handle* pointer will be invalid. Its counterpart function is the function *sv_render_open()*.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

int sv_render_end (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*)

This function closes the render context. Its counterpart function is the function *sv_render_begin()*.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Handle to the render context. After this function call the *sv_render_context* handle will be invalid.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

Note:

After using this function you have to call the function *sv_render_free()* to make the allocated buffers available again.

See also:

The function *sv_render_reuse()*.

int sv_render_issue (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, sv_overlapped * *poverlapped*)

This function starts the render operation. In case a user allocated *sv_overlapped* structure is set, it will return immediately and you have to wait for the render operation to finish with the function *sv_render_ready()*. If such a structure was not set, this function will block until the render process is complete, i.e. a wait with *sv_render_ready()* is not required.

After rendering you can transfer the resulting data via a DMA transfer to the system memory. Then you may reset the render context with the function *sv_render_reuse()* or close it with the function *sv_render_end()*.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Handle to the render context.

poverlapped – Pointer to a user allocated *sv_overlapped* structure to enable an asynchronous operation. With this you have to wait for the render operation to finish with the function *sv_render_ready()*. If set to `NULL`, this function is performed synchronously and blocks until the render process is complete.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

int sv_render_open (sv_handle * *sv*, sv_render_handle ** *pprender*)

This function opens a global Render API handle. You will need this handle for all following Render API functions. When the processing has been finished completely, you have to close the handle with the function *sv_render_close()*.

Parameters:

sv – Handle returned from the function *sv_open()*.

pprender – Pointer to the returned render handle.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

Note:

You can open only one handle at a time, but you can use this handle in more than one render context (see the function *sv_render_begin()*).

This function returns an error if used with the wrong hardware or a missing license.

When calling this function without an output FIFO in use, it will disable the outputs on the DVS video board. This will save internal memory bandwidth to ensure full render performance for render-only processing.

int sv_render_option_get (sv_handle * *sv*, sv_render_handle * *prender*, int *option*, int * *pvalue*)

This function retrieves an `SV_RENDER_OPTION_<xxx>` value.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

option – `SV_RENDER_OPTION_<xxx>` define. For possible defines see the corresponding chapters in this reference guide.

pvalue – Pointer to the value to be returned.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

See also:

The function *sv_render_option_set()*.

int sv_render_option_set (sv_handle * *sv*, sv_render_handle * *prender*, int *option*, int *value*)

This function sets an `SV_RENDER_OPTION_<xxx>` value.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

option – `SV_RENDER_OPTION_<xxx>` define. For possible defines see the corresponding chapters in this reference guide.

value – Value that should be set for *option*.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

See also:

The function *sv_render_option_get()*.

int sv_render_ready (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, int *timeout*, sv_overlapped * *poverlapped*)

This function has to be called in conjunction with the function *sv_render_issue()* if an *sv_overlapped* structure is used. It performs a conditional wait until the render operation is finished, meaning it waits for the event in the *sv_overlapped* structure and when occurred returns immediately.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Handle to the render context.

timeout – Currently not used. In the future it may provide a timeout in microseconds to wait until the buffer is ready, `-1` will be infinite.

poverlapped – Pointer to the *sv_overlapped* structure that has been initialized with the function *sv_render_issue()*.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_NOTREADY`.

int sv_render_reuse (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*)

This function resets the *sv_render_context* handle. Afterwards the *sv_render_context* handle will provide the same state as after the function *sv_render_begin()*. With this you do not have to delete and re-allocate an *sv_render_context* handle if you want to render more than one frame. You can also use the same *sv_render_image* handles in this render context again if the buffer size has not changed, i.e. you do not have to re-allocate them with the function *sv_render_malloc()*.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Handle to the render context.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

API – Render API – *Memory Functions*

Detailed Description

The functions described in this chapter allow you to manage the memory on the DVS video board for the Render API, and to transfer the buffer from the board to the system memory or vice versa.

Data Structures

- struct sv_render_bufferlayout

Functions

- int sv_render_dma (sv_handle *sv, sv_render_handle *prender, int btocard, sv_render_image *pimage, void *buffer, int bufferoffset, int transfersize, sv_overlapped *poverlapped)
- int sv_render_dmaex (sv_handle *sv, sv_render_handle *prender, int btocard, sv_render_image *pimage, char *memoryaddr, int memorysize, int memoryoffset, int memorylineoffset, int cardoffset, int cardlineoffset, int linesize, int linecount, int spare, sv_overlapped *poverlapped)
- int sv_render_free (sv_handle *sv, sv_render_handle *prender, sv_render_image *pimage)
- int sv_render_malloc (sv_handle *sv, sv_render_handle *prender, sv_render_image **ppimage, int version, int size, sv_render_bufferlayout *pstorage)
- int sv_render_memory_info (sv_handle *sv, sv_render_handle *prender, int *ptotal, int *pfree, int *pfreeblock)

Function Documentation

int sv_render_dma (sv_handle * *sv*, sv_render_handle * *prender*, int *btocard*, sv_render_image * *pimage*, void * *buffer*, int *bufferoffset*, int *transfersize*, sv_overlapped * *poverlapped*)

This function performs a DMA (read or write) to a buffer in the storage of the DVS video board or to a specific memory address in the CPU memory.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

btocard – If you want to transfer to the video device's memory, set this parameter to `TRUE`. In case you want to transfer to the CPU memory, set it to `FALSE`.

pimage – Handle to the board buffer.

buffer – Memory address in the CPU memory.

bufferoffset – Offset in the CPU memory. This value is relative to *buffer*.

transfersize – Size of the buffer at *buffer*.

poverlapped – Overlapped structure for I/O operations. If this is set to `NULL`, a normal synchronous DMA transfer is done, otherwise the transfer will be performed asynchronously. On UNIX systems this parameter should be `NULL`.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

Note:

The buffer in the CPU memory has to be aligned to the needed DMA alignment of the DVS video board. You can query the DMA alignment with the function *sv_query()* and the define `SV_QUERY_DMAALIGNMENT`.

See also:

The function *sv_render_dmaex()*.

int sv_render_dmaex (sv_handle * *sv*, sv_render_handle * *prender*, int *btocard*, sv_render_image * *pimage*, char * *memoryaddr*, int *memorysize*, int *memoryoffset*, int *memorylineoffset*, int *cardoffset*, int *cardlineoffset*, int *linesize*, int *linecount*, int *spare*, sv_overlapped * *poverlapped*)

This function performs a DMA (read or write) to a buffer in the storage of the DVS video board or to a specific memory address in the CPU memory. Compared to the function *sv_render_dma()* it offers more advanced DMA capabilities such as a cut-out and/or stride in the system memory.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

btocard – If you want to transfer to the video device's memory, set this parameter to `TRUE`. In case you want to transfer to the CPU memory, set it to `FALSE`.

pimage – Handle to the board buffer.

memoryaddr – Memory address in the CPU memory.

memorysize – Size of the buffer at *memoryaddr*.

memoryoffset – Offset in the CPU memory. This value is relative to *memoryaddr*.

memorylineoffset – Line offset in the CPU memory in bytes (from the beginning of a line to the beginning of the next line).

cardoffset – Offset in the video device memory.

cardlineoffset – Line offset in the video device memory in bytes (from the beginning of a line to the beginning of the next line).

linesize – Size of each line.

linecount – Number of lines.

spare – Currently not used. It has to be set to zero (`0`).

poverlapped – Overlapped structure for I/O operations. If this is set to `NULL`, a normal synchronous DMA transfer is done, otherwise the transfer will be performed asynchronously. On UNIX systems this parameter should be `NULL`.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

Note:

The buffer in the CPU memory has to be aligned to the needed DMA alignment of the DVS video board. You can query the DMA alignment with the function *sv_query()* and the define `SV_QUERY_DMAALIGNMENT`.

See also:

The function *sv_render_dma()*.

int sv_render_free (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_image * *pimage*)

This function frees the memory on the DVS video board that was previously allocated by its counterpart function *sv_render_malloc()*. It should be called before ending a render context with the function *sv_render_end()*.

By setting *sv_render_image.bufferid* to zero (`0`), this function will free only the local system memory consumed by the *sv_render_image* handle, but not the virtual buffer on the DVS video board. For further information about this please refer to chapter API – FIFO API – Usage with the Render API.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pimage – Handle to the buffer allocated on the hardware.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

int sv_render_malloc (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_image ** *ppimage*, int *version*, int *size*, sv_render_bufferlayout * *pstorage*)

This function allocates memory on the DVS video board for a buffer to be used for rendering. Its counterpart function is the function *sv_render_free()*.

If your data is JPEG2000 compressed, you have to use version 2 of the structure *sv_render_bufferlayout*. With uncompressed data version 1 has to be used.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

ppimage – Handle to the buffer allocated on the hardware.

version – Version of the structure that will be used. Set to one (`1`) if *pstorage->v1*.

size – Size of the structure *sv_render_bufferlayout*.

pstorage – Size of the buffer to be allocated.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

Note:

If the size of the source image differs from the size of the target image, an automatic scaling will be performed. For further information about this see the function *sv_render_push_scaler()*.

int sv_render_memory_info (sv_handle * *sv*, sv_render_handle * *prender*, int * *ptotal*, int * *pfree*, int * *pfreeblock*)

This function returns information about the DVS video board memory available for the Render API. All values are in megabyte (MB).

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

ptotal – If this pointer is set, it will be filled with the total amount of memory available for the Render API.

pfree – If this pointer is set, it will be filled with the total amount of memory that is free.

pfreeblock – If this pointer is set, it will be filled with the size of the largest free block.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

API – Render API – Stack Operator Functions

Detailed Description

Operators work on one or more (`1..n`) images on the stack and perform a specific operation on the image buffers. This operation can be configured via settings elements (`0..n`). When using settings elements, they have to be pushed on the stack preceding the operator (i.e. between image and operator). The resulting image is then pushed back on the stack. Currently it is possible to push one operator element per stack only. Pushing multiple operators of the same type on the stack may be implemented in a future version of this API.

Functions

- int sv_render_push_render (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, sv_render_image *pimage)

Function Documentation

int sv_render_push_render (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, sv_render_image * *pimage*)

This function pushes the render operator on the render stack. When the render operator is processed by the function *sv_render_issue()*, it will pop the preceding image and all corresponding settings from the stack, after that it will apply all settings to the image and push the resulting image back to the stack.

Additionally, the render operator will save the image to a specific memory position on the video board. With this you can directly transfer the buffer via a DMA transfer from the video board memory to the system RAM and you do not have to pop the image from the stack specifically.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Return handle to the render context.

pimage – Handle to the buffer on the DVS video device.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

API – Render API – Stack Image Functions

Detailed Description

Images are actually image buffers which are used by operators to create a new image buffer where appropriate. It is possible to push multiple (`0..n`) images on the stack.

Functions

- int sv_render_push_image (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, sv_render_image *pimage)

Function Documentation

int sv_render_push_image (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, sv_render_image * *pimage*)

This function pushes the video data of a source image on the render stack for processing.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Return handle to the render context.

pimage – Handle to the buffer on the DVS video device where the source image is stored.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

Note:

Currently only one stack level is allowed, because there is only the render operator available to process a single image. In the future it may be possible to push more than one image to the stack.

API – Render API – Stack Settings Functions

Detailed Description

Settings are configuration elements for an operator which have to be pushed on the stack before pushing the operator itself. They are passive elements that will be evaluated by the succeeding operator. You can push multiple (`0..n`) settings elements on the stack and they can be pushed in any sequence you want. The final processing sequence is determined by the DVS hardware (see section Hardware Render Pipelines).

Data Structures

- struct sv_render_1dlut
- struct sv_render_3dlut
- struct sv_render_jpeg2000_decode
- struct sv_render_matrix
- struct sv_render_scaler

Functions

- int sv_render_push_1dlut (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, int version, int size, sv_render_1dlut *pvalue)
- int sv_render_push_3dlut (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, int version, int size, sv_render_3dlut *pvalue)
- int sv_render_push_jpeg2000_decode (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, sv_render_image *pimage, int version, int size, sv_render_jpeg2000_decode *pvalue)
- int sv_render_push_matrix (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, int version, int size, sv_render_matrix *pvalue)
- int sv_render_push_scaler (sv_handle *sv, sv_render_handle *prender, sv_render_context *pcontext, sv_render_image *pdest, int version, int size, sv_render_scaler *pvalue)

Function Documentation

int sv_render_push_1dlut (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, int *version*, int *size*, sv_render_1dlut * *pvalue*)

This function pushes a 1D LUT setting for processing on the render stack. The 1D LUT will be enabled for the preceding image and applied by the subsequent operator.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Return handle to the render context.

version – Version of the structure that will be used. Set to one (`1`) for *pvalue->v1*.

size – Size of the structure *sv_render_1dlut*.

pvalue – Pointer to the LUT data to be used.

Parameters for *pvalue->flags* (Flags):

- `SV_RENDER_LUTFLAGS_NOTLINEAR` – The buffer contains a 1D LUT (RGBA32) with non-linear nodes. Non-linear nodes mean that the first LUT entries (50 %), e.g. 512 entries for a 10-bit LUT, describe the LUT with a higher resolution in step sizes of `1/65536`. The remaining 512 entries describe the LUT in step sizes of `1/512`. The entries `512..515` are not evaluated because they are already covered by the entries `0..511`. Such an LUT is especially useful for curves with a high gradient near the zero point, e.g. `x^gamma` (for `gamma < 1`).

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

int sv_render_push_3dlut (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, int *version*, int *size*, sv_render_3dlut * *pvalue*)

This function pushes a 3D LUT setting for processing on the render stack. The 3D LUT will be enabled for the preceding image and applied by the subsequent operator.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Return handle to the render context.

version – Version of the structure that will be used. Set to one (`1`) for *pvalue->v1*.

size – Size of the structure *sv_render_3dlut*.

pvalue – Pointer to the LUT data to be used.

Data Layout of a 3D LUT:

The native data layout of a 3D LUT is GBR16. There are 17*17*17 entries. The size of the LUT data normally is GBR * 2 bytes * entries (i.e. 3 * 2 * 17^3), resulting in 29478 bytes for a 16-bit 3D LUT. The components are interleaved:

```
G00B00R00 G00B00R01 G00B00R02 ... G00B00R16
G00B01R00 G00B01R01 G00B01R02 ... G00B01R16
 ...
G00B16R00 G00B16R01 G00B16R02 ... G00B16R16
G01B00R00 G01B00R01 G01B00R02 ... G01B00R16
 ...
G16B16R00 G16B16R01 G16B16R02 ... G16B16R16
```

For performance reasons, the function always expects a 32768-bytes LUT buffer. However, only the first 29478 bytes will be used, and subsequent bytes will be disregarded. In any case, the values always range from zero to 65535 (`0..65535`).

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

int sv_render_push_jpeg2000_decode (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, sv_render_image * *pimage*, int *version*, int *size*, sv_render_jpeg2000_decode * *pvalue*)

This function pushes the JPEG2000 decompression setting for processing on the stack. The JPEG2000 decompression will be enabled for the preceding image and applied by the subsequent operator.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Return handle to the render context.

pimage – Handle to the buffer on the DVS video device.

version – Version of the structure that will be used. Set to one (`1`) for *pvalue->v1*.

size – Size of the structure *sv_render_jpeg2000_decode*.

pvalue – Pointer to the decoding parameters.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

int sv_render_push_matrix (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, int *version*, int *size*, sv_render_matrix * *pvalue*)

This function pushes a matrix setting for processing on the render stack. The matrix will be enabled for the preceding image and applied by the subsequent operator.

With more than one matrix in your render pipeline at least version 2 of the structure *sv_render_matrix* is required. In this version of the structure you can set a matrix ID, the complete matrix values and the in- and out-offsets. If only a simple color space conversion should be performed, you can use version 3 of the structure, where you can set a source and destination color space. Then, during rendering the DVS video board driver calculates the correct matrix values automatically.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Return handle to the render context.

version – Version of the structure that will be used. Set to one (`1`) if *pvalue->v1*.

size – Size of the structure *sv_render_matrix*.

pvalue – Pointer to the matrix data to be used.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

int sv_render_push_scaler (sv_handle * *sv*, sv_render_handle * *prender*, sv_render_context * *pcontext*, sv_render_image * *pdest*, int *version*, int *size*, sv_render_scaler * *pvalue*)

This function pushes the scaler setting for processing on the render stack. The scaling will be enabled for the preceding image and applied by the subsequent operator.

This setting has to be used only when you want to scale within the destination image (e.g. to apply letterboxing): An automatic scaling will be performed already if the destination sizes of the destination image differ from the sizes of the source image allocated with the function *sv_render_malloc()*. With this, an explicit setting for a scaling may not be necessary.

There are limitations regarding an explicit or automatic scaling:

1. When an automatic scaling is performed, only a downscaling is possible.
2. You can only scale to an x-size of 4096 pixels or less.

Parameters:

sv – Handle returned from the function *sv_open()*.

prender – Handle to the render functions returned from the function *sv_render_open()*.

pcontext – Return handle to the render context.

pdest – Handle to the destination buffer on the DVS video device.

version – Version of the structure that will be used. Set to one (`1`) if *pvalue->v1*.

size – Size of the structure *sv_render_scaler*.

pvalue – Pointer to the scaler parameters.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

See also:

The function *sv_render_malloc()*.

API – FIFO API – Usage with the Render API

Detailed Description

It is possible to use the FIFO API together with the Render API. Then you can, for example, apply the hardware accelerated JPEG2000 decompression of the Render API and afterwards display the rendered image directly on the digital video output(s) via a virtual video buffer for the FIFO API.

Before using the FIFO API together with the Render API you have to set a FIFO memory mode (see the function *sv_fifo_memorymode()*). It has to be set prior to initializing an output FIFO and offers you several possibilities:

- By setting the FIFO API to `SV_FIFO_MEMORYMODE_SHARE_RENDER` the FIFO will use the internal dynamic RAM allocator to allocate buffer space, i.e. it uses exactly the same allocator as the function *sv_render_malloc()*. With this you can, for example, transfer the buffer after rendering back to the system memory for further processing, and afterwards to the FIFO API buffer on the DVS video board for an output.
- It is also possible to transfer the rendered images directly to the FIFO API without bothering the system with a further DMA transfer, thereby saving a lot of performance. With this the FIFO API does not need a ring buffer of its own (`SV_FIFO_MEMORYMODE_FIFO_NONE`), because it will display the buffer directly from its render position (virtual video buffer). In this mode it is important that you deactivate the DMA transfer of the FIFO (i.e. set *bdma* to two (`2`) in *sv_fifo_init()*). Furthermore, by setting *pbuffer->storage.bufferid* as shown in the following example, the FIFO API will free the render buffer automatically as soon as the image has been displayed. Nevertheless, you still have to call the function *sv_render_free()* for this particular buffer as well to free the local system memory which is consumed by the *sv_render_image* handle. Then, to force *sv_render_free()* to free the local system memory only and not the virtual render buffer, you have to set *bufferid* to zero (`0`) before calling it.

```
int res = SV_OK;
sv_render_image virtual_render_image;
sv_fifo_memory memory;
sv_fifo_buffer *pbuffer;

memory.mode = SV_FIFO_MEMORYMODE_FIFO_NONE;
memory.size = 0; // default
memory.pad = 0;

res = sv_fifo_memorymode(sv, &memory);
if(res != SV_OK) {
  ... // Appropriate error handling
}

// Set the bdma parameter to 2 (i.e. to DMA OFF)
res = sv_fifo_init(sv, &pfifo, 0, 0, 2, 0, 0);
if(res != SV_OK) {
  ...
}

res = sv_fifo_getbuffer(sv, pfifo, &pbuffer, 0, 0);
if(res != SV_OK) {
  ...
}

// Put the render image into the pbuffer structure
pbuffer->dma.addr = (char*)virtual_render_image->bufferoffset;
pbuffer->dma.size = virtual_render_image->buffersize;

// If set, an automatic freeing of the internal render buffer will be performed,
// otherwise it won't:
pbuffer->storage.bufferid    = virtual_render_image->bufferid;
```

```
pbuffer->storage.xsize       = virtual_render_image->xsize;
pbuffer->storage.ysize       = virtual_render_image->ysize;
pbuffer->storage.storagemode = virtual_render_image->storagemode;
pbuffer->storage.matrixtype  = virtual_render_image->matrixtype;
pbuffer->storage.lineoffset  = virtual_render_image->lineoffset;
pbuffer->storage.dataoffset  = virtual_render_image->dataoffset;

// Transfer the rendered image to the FIFO API
res = sv_fifo_putbuffer(sv, pfifo, pbuffer, 0);
if(res != SV_OK) {
  ...
}
```

However, when using the FIFO API together with the Render API, some conditions should be observed:

1. You can use one global *sv_handle* pointer from the function *sv_open()* for your application (FIFO API and Render API). But in case two or more are required, you can open separate *sv_handle* pointers by opening different ports with the function *sv_openex()*, for example, `SV_OPENTYPE_VOUTPUT` and `SV_OPENTYPE_RENDER`. For more information please refer to the description of the function *sv_openex()* in the "DVS Software Development Kit" reference guide.
2. When using the mode `SV_FIFO_MEMORYMODE_SHARE_RENDER`, some data rate limitations on the PCIe bus have to be observed, because every DMA transfer causes a specific amount of load. In this mode the FIFO API as well as the Render API initiate each DMA transfers. Therefore, before using it check your application whether it is possible to transfer the intended amount of data in the available time. The data rate depends on the size of the buffer and the frequency of the video raster.

This chapter describes the defines and functions that are required to use the FIFO API together with the Render API.

Data Structures

- struct sv_fifo_memory

Defines

- #define SV_QUERY_FIFO_MEMORYMODE

Functions

- int sv_fifo_memorymode (sv_handle *sv, sv_fifo_memory *pmemory)

Define Documentation

#define SV_QUERY_FIFO_MEMORYMODE

This define returns the current memory mode set for the FIFO. See the function *sv_fifo_memorymode()*.

Function Documentation

int sv_fifo_memorymode (sv_handle * *sv*, sv_fifo_memory * *pmemory*)

This function sets the memory mode for the FIFO API. It is only necessary if you want use the FIFO API together with the Render API. The parameters listed below can be set in the element *mode* of the structure *sv_fifo_memory*. To set a memory mode all FIFOs have to be closed.

Parameters:

sv – Handle returned from the function *sv_open()*.

pmemory – Handle to the *sv_fifo_memory* structure.

Parameters for *pmemory->mode* (Flags):

- `SV_FIFO_MEMORYMODE_FIFO_ALL` – The complete memory of the DVS video board will be used for the FIFO.
- `SV_FIFO_MEMORYMODE_DEFAULT` – Default memory mode, i.e. the same as `SV_FIFO_MEMORYMODE_FIFO_ALL`.
- `SV_FIFO_MEMORYMODE_FIFO_NONE` – The FIFO does not have any memory of its own.
- `SV_FIFO_MEMORYMODE_SHARE_RENDER` – The memory will be shared between FIFO and Render API. In this mode the *sv_fifo_memory.size* value is evaluated. By setting it to zero (`0`) a third of the available space will be dedicated to the FIFO automatically.

Returns:

If the function succeeds, it returns `SV_OK`. Otherwise it will return the error code `SV_ERROR_<xxx>`.

Note:

The mode `SV_FIFO_MEMORYMODE_DEFAULT` cannot be used together with the Render API.

See also:

The define `SV_QUERY_FIFO_MEMORYMODE`.

DVS SDK Data Structure Documentation

sv_fifo_memory Struct Reference

Detailed Description

The following describes the structure *sv_fifo_memory* used by the function *sv_fifo_memorymode()*.

```
typedef struct {
  int mode;       // Memory mode (SV_FIFO_MEMORYMODE_<xxx> flags). See the function
                  // sv_fifo_memorymode().
  int size;       // Memory size, zero (0) is default.
  int pad[16];    // Reserved for future use. Set to zero (0).
} sv_fifo_memory;
```

sv_render_1dlut Struct Reference

Detailed Description

The following details the buffer structure *sv_render_1dlut* used by the function *sv_render_push_1dlut()*.

```
typedef union {
  struct {
    void * plut;              // Pointer to the buffer containing the LUT data.
    int    size;              // Size of the LUT data, e.g. RGBA * 4 bytes * entries
                              // -> 4 * 4 * 1024 or 4 * 4 * 1025 for a 10-bit LUT.
                              // The components are non-interleaved, i.e.
                              // "R0R1R2...G0G1G2...B0B1B2...A0A1A2...".
    int    lutid;             // LUT ID in case there are multiple LUTs available on
                              // the data path (0..n).
    int    flags;             // Flags for LUT processing (see
                              // SV_RENDER_LUTFLAGS_<xxx> in function
                              // sv_render_push_1dlut()).
    int    spare[4];          // Reserved for future use.
  } v1;                       // Version 1 of the structure.
} sv_render_1dlut;
```

sv_render_3dlut Struct Reference

Detailed Description

The following details the buffer structure *sv_render_3dlut* used by the function *sv_render_push_3dlut()*.

```
typedef union {
  struct {
    void * plut;              // Pointer to the buffer containing the LUT data.
    int    size;              // Size of the LUT data. See function
                              // sv_render_push_3dlut() for details.
    int    lutid;             // LUT ID in case there are multiple 3D LUTs available
                              // on the data path (0..n).
    int    flags;             // Reserved for future use.
    int    spare[4];          // Reserved for future use.
  } v1;                       // Version 1 of the structure.
} sv_render_3dlut;
```

sv_render_bufferlayout Struct Reference

Detailed Description

The following describes the buffer structure *sv_render_bufferlayout* used by the function *sv_render_malloc()*.

```
typedef union {
  struct {
    int xsize;                // X-size of uncompressed data.
    int ysize;                // Y-size of uncompressed data.
    int storagemode;          // Storage mode (SV_MODE_<xxx>).
    int lineoffset;           // Offset from line to line (default is zero (0)).
    int matrixtype;           // Color space (SV_MATRIXTYPE_<xxx>).
    int dataoffset;           // Offset to the start of the data.
  } v1;                       // Version 1 of the structure used for uncompressed
                              // buffers.
  struct {
    int xsize;                // X-size of compressed data.
    int ysize;                // Y-size of compressed data.
    int buffersize;           // Size of the compressed data.
    int dataoffset;           // Offset to the start of the data.
    int storagemode;          // Storage mode (SV_MODE_<xxx>).
    int matrixtype;           // Color space (SV_MATRIXTYPE_<xxx>).
  } v2;                       // Version 2 of the structure used for compressed
                              // buffers.
} sv_render_bufferlayout;
```

sv_render_jpeg2000_decode Struct Reference

Detailed Description

The following describes the buffer structure *sv_render_jpeg2000_decode* used by the function *sv_render_push_jpeg2000_decode()*.

```
typedef union {
  struct {
    void *  addr;             // Address of the JPEG2000 source data (incl.
                              // plaintext).
    int     size;             // Size of the JPEG2000 source data.
    int     encryption;       // Encryption mode (SV_ENCRYPTION_<xxx>).
    int     keyid;            // Decryption key ID.
    int     payload;          // Amount of data (incl. plaintext and padding).
    int     plaintext;        // Plaintext offset.
    int     sourcelength;     // Original size of the non-encrypted data.
  } v1;                       // Version 1 of the structure.
} sv_render_jpeg2000_decode;
```

sv_render_matrix Struct Reference

Detailed Description

The following describes the buffer structure *sv_render_matrix* used by the function *sv_render_push_matrix()*.

```
typedef union {
  struct {
    double matrix[10];        // Matrix coefficients (3 * 3 plus 1 for alpha).
    double offset[4];         // Matrix offsets (one for each component).
  } v1;                       // Version 1 of the structure.
  struct {
    double matrix[10];        // Matrix coefficients (3 * 3 plus 1 for alpha).
    double inoffset[4];       // Matrix in-offsets (one for each component).
    double outoffset[4];      // Matrix out-offsets (one for each component).
    int matrixid;             // Matrix position within processing pipeline (0..n).
  } v2;                       // Version 2 of the structure.
  struct {
    int matrixtype_source;    // Source color space (SV_MATRIXTYPE_<xxx>).
    int matrixtype_dest;      // Destination color space (SV_MATRIXTYPE_<xxx>).
    int matrixid;             // Matrix position within processing pipeline (0..n).
  } v3;                       // Version 3 of the structure.
} sv_render_matrix;
```

sv_render_scaler Struct Reference

Detailed Description

The following describes the buffer structure *sv_render_scaler* used by the function *sv_render_push_scaler()*.

```
typedef union {
  struct {
    int xsize;                // Destination x-size.
    int ysize;                // Destination y-size.
    int xoffset;              // Currently not used, set to zero (0).
    int yoffset;              // Currently not used, set to zero (0).
  } v1;                       // Version 1 of the structure.
  struct {
    int xsize;                // Destination x-size.
    int ysize;                // Destination y-size.
    int xoffset;              // Currently not used, set to zero (0).
    int yoffset;              // Currently not used, set to zero (0).
    int sharpness;            // Sharpness, valid from -0xffff to +0xffff, default is
                              // zero (0).
  } v2;                       // Version 2 of the structure.
} sv_render_scaler;
```

Index