

# **PlayStation®Edge Post Library Reference**

# Table of Contents

<b>Preface.....</b>	<b>5</b>
About This Document.....	6
<b>Data Types.....</b>	<b>7</b>
EdgePostSourceTile.....	8
EdgePostProcessStage .....	10
EdgePostWorkload.....	12
EdgePostTileInfo .....	13
EdgePostSpuConfig .....	14
EdgePostImage.....	15
EdgePostMlaaContext .....	16
EdgePostMlaaTaskParameters.....	17
EdgePostMlaaMemoryLayout .....	19
EdgePostInplaceTransposeMemoryLayout .....	20
EdgePostInplaceTransposePostOpFunction .....	21
<b>SPU Functions .....</b>	<b>22</b>
edgePostSetSpuConfig .....	23
edgePostRunWorkload .....	24
edgePostMlaaPass .....	26
edgePostInplaceTransposePass .....	27
edgePostTransposeInPlace .....	28
<b>SPU Processing Functions.....</b>	<b>30</b>
edgePostDownsample8 .....	31
edgePostDownsample16 .....	32
edgePostDownsampleF .....	33
edgePostNearestDownsample.....	34
edgePostDownsampleFloatMin .....	35
edgePostDownsampleFloatMax .....	36
edgePostUpsample8 .....	37
edgePostUpsample16 .....	38
edgePostUpsampleF .....	39
edgePostGauss7x1_8.....	40
edgePostGauss7x1F.....	41
edgePostGauss1x7_8.....	42
edgePostGauss1x7F.....	43
edgePostBloomCapture8 .....	44
edgePostBloomCaptureFX16 .....	45
edgePostTonemapFX16.....	46
edgePostAvgLuminanceFX16.....	47
edgePostModulate8 .....	48
edgePostModulateFX16.....	49
edgePostConstantModulate8 .....	50
edgePostAddSat8 .....	51
edgePostAddSatFX16.....	52
edgePostBlend8 .....	53

edgePostBlendFX16 .....	54
edgePostPremultiplyFX16 .....	55
edgePostCombine .....	56
edgePostExtractDepth .....	57
edgePostFloatToGrayscale .....	58
edgePostArgb8ToFloats .....	59
edgePostFloatsToArgb8 .....	60
edgePostFX16ToFloats .....	61
edgePostFX16ToArgb8 .....	62
edgePostFP16LoToFloats .....	63
edgePostFP16HiToFloats .....	64
edgePostFP16ToFloats .....	65
edgePostFloatsToFP16 .....	66
edgePostFX16ToFP16 .....	67
edgePostLogLuvToFloats .....	68
edgePostLogLuvToFX16 .....	69
edgePostLogLuvToArgb .....	70
edgePostFloatsToLogLuv .....	71
edgePostFX16ToLogLuv .....	72
edgePostFloatsToLuv .....	73
edgePostFX16ToLuv .....	74
edgePostLuvToFloats .....	75
edgePostLuvToFX16 .....	76
edgePostLuvToArgb .....	77
edgePostFP16LuvToFloats .....	78
edgePostFP16LuvToFX16 .....	79
edgePostFloatsToFP16Luv .....	80
edgePostFX16ToFP16Luv .....	81
edgePostDof .....	82
edgePostDof_FX16 .....	83
edgePostInitializeDofInputBuffer .....	84
edgePostExtractNearFuzziness .....	85
edgePostExtractFarFuzziness .....	86
edgePostMotionblur .....	87
edgePostMotionblur_FX16 .....	88
edgePostMakeMaskFromFloats .....	89
edgePostApplyMaskFX16 .....	90
<b>PPU Functions .....</b>	<b>91</b>
edgePostInitializeWorkload .....	92
edgePostIsWorkloadFinished .....	93
edgePostStallForWorkload .....	94
edgePostSelectTileSize .....	95
edgePostSetupStage .....	96
edgePostMlaaInitializeContext .....	98
edgePostMlaaDestroyContext .....	99
edgePostMlaaPrepareWithRelativeThreshold .....	100
edgePostMlaaKickTasks .....	101
edgePostMlaaWait .....	102

---

<b>SPU Callback Functions .....</b>	<b>103</b>
EdgePostPollCallback .....	104
EdgePostStageEnterCallback .....	105
EdgePostStageExitCallback .....	106
EdgePostTileCallback .....	107
<b>Constants .....</b>	<b>108</b>
Return Codes .....	109

# Preface

---

# About This Document

---

## Purpose

This document provides an API reference for the Edge Post component of the Edge library. Use this component to perform image post-processing on SPU.

## Audience and Prerequisites

This document was written for PlayStation®3 developers who want to write high-performance applications for the PlayStation®3. It is assumed that such developers have familiarity with the following:

- C and C++
- PlayStation®3 hardware
- SCE standard library functions

## Related Documentation and Other Resources

### Edge Library

In combination with this reference, the following documents provide complete usage and reference information about the Edge library:

- *PlayStation®Edge Library Overview*
- *PlayStation®Edge Geometry Library: Quick Start*
- *PlayStation®Edge Geometry Library Reference*
- *PlayStation®Edge Geometry Library for Offline Tool: Reference*
- *PlayStation®Edge Animation Library Reference*
- *PlayStation®Edge Animation Library for Offline Tool: Reference*
- *PlayStation®Edge Zlib Library Reference*
- *PlayStation®Edge LZMA Library Reference*
- *PlayStation®Edge LZO Library Reference*
- *PlayStation®Edge DXT Library Reference*

### SPA and EdgePostFilterGen Tools

The *SPU Pipelining Assembler (SPA) User's Guide*, included with the Edge package, describes the SPA, an assembly optimization tool.

The *EdgePostFilterGen User's Guide*, also included with the Edge package, describes EdgePostFilterGen, a command-line utility for generating SPA-optimized loops for simple image kernel operations (such as Gaussian image filters). These loops can be used in Edge Post. The output is in the form of an SPA file ready to be fed to the SPA tool.

## Typographic Conventions

This document uses the following typographic conventions:

Convention	Meaning
fixed-width font	Indicates programming code and literals, such as processing instructions, register names, data types, events, and file names. Also indicates function, structure, and macro names.
<b>fixed-width font + bold</b>	In structure/function definitions only, indicates names of structures and functions.
<i>fixed-width font + italics</i>	Indicates arguments, parameters, and variables.
<u>blue + underlined text</u>	Indicates a hyperlink (blue displays in color printers or online only).

# Data Types

# EdgePostSourceTile

Structure that describes an input image tile configuration.

## Definition

```
#include <edgepost.h>
struct EdgePostSourceTile
{
    uint32_t addressEa;
    uint32_t pitch;
    uint16_t width;
    uint16_t height;
    uint8_t borderWidth;
    uint8_t borderHeight;
    uint8_t bytesPerPixel;
    uint8_t multiplier:6;
    uint8_t type:2;
} __attribute__((__aligned__(16)));
```

## Members

<i>addressEa</i>	Effective address of the start of the image in main memory
<i>pitch</i>	Pitch of the image
<i>width</i>	Width in pixels for one tile inside the image
<i>height</i>	Height in pixels for one tile inside the image
<i>borderWidth</i>	Width of the required border along left and right edges of the tile
<i>borderHeight</i>	Height of the required border along upper and bottom edges of the tile
<i>bytesPerPixel</i>	Size of the pixel format in bytes
<i>multiplier</i>	Multiplier factor used when calculating local-store occupancy for one tile
<i>type</i>	Type of tile: input or output

## Description

This is the structure used to describe an input image and its tile configuration.

*addressEa* stores the 16-byte aligned starting address of the entire image. *pitch* is the pitch of the image, which is usually the width of the entire image multiplied by the number of bytes per pixel.

*width* and *height* store the chosen tile size for the image; they are not the dimensions for the entire image.

*borderWidth* and *borderHeight* are the size required for the borders around the tile that needs to be present in local store when the tile is processed; tile borders are required every time the result of the computation of one pixel is dependent on its surrounding pixels. You should specify a border large enough to be able to access the farthest pixel from your center; for example to execute a horizontal 9x1 gauss blur you will need a border of at least 4 pixels.

*borderWidth* is specified in pixels, and its real size in bytes ( $borderWidth * bytesPerPixel$ ) must be 16-byte aligned. *borderHeight* is specified in terms of number of scanlines (above and below the tile).

Note that the real pitch of a tile can be calculated as  $(borderWidth * 2 + width) * bytesPerPixel$ . This value must be 16-byte aligned and never bigger than 16 KB, because each tile line is transferred using a single DMA operation.

*bytesPerPixel* stores the size in bytes for each pixel (usually 4 for an ARGB image).

*multiplier* is a factor used to calculate the real occupancy of the tile in SPU local store. It can be used to over-allocate enough space to be able to apply certain transformations to the tile before using it.



The tile occupancy in SPU local store can be computed as follows:

$(width + borderWidth * 2) * bytesPerPixel * multiplier * (height + borderHeight * 2)$

Finally, type tells whether the tile will be treated as an input or output tile.

#### **See Also**

---

[EdgePostProcessStage](#)

# EdgePostProcessStage

Structure that describes a stage in an effect chain.

## Definition

```
#include <edgepost.h>
struct EdgePostProcessStage
{
    uint8_t numTileX;
    uint8_t numTileY;
    uint8_t pad;
    uint8_t flags;
    uint32_t rsxLabelAddress;
    uint32_t rsxLabelValue;
    uint32_t effectCodeEa;
    uint32_t effectCodeSize;
    uint32_t effectCodeDmaSize;
    uint32_t stageParametersEa;
    uint32_t stageParametersSize;
    uint32_t user0;
    uint32_t user1;
    uint32_t user2;
    uint32_t user3;
    union {
        uint32_t userDataI[4];
        float userDataF[4];
    };
    EdgePostSourceTile sources[ EDGE_POST_MAX_SOURCE_TILES ];
} __attribute__((__aligned__(16)));
```

## Members

<i>numTileX</i>	Number of tiles per row
<i>numTileY</i>	Number of tile rows
<i>pad</i>	Structure padding (currently unused)
<i>flags</i>	Generic flags
<i>rsxLabelAddress</i>	Optional address of an RSX™ label
<i>rsxLabelValue</i>	Optional value to be used to update the RSX™ label
<i>effectCodeEa</i>	Optional effective address of the effect code
<i>effectCodeSize</i>	Amount of memory to be reserved for the effect code
<i>effectCodeDmaSize</i>	Size of the DMA transfer of the effect code
<i>stageParametersEa</i>	Effective address of stage parameter area
<i>stageParametersSize</i>	Size of the stage parameter area
<i>user0</i>	User data
<i>user1</i>	User data
<i>user2</i>	User data
<i>user3</i>	User data
<i>userDataI</i>	User data
<i>userDataF</i>	User data
<i>sources</i>	An array of tile descriptors

## Description

This is Edge Post's fundamental structure since it holds all the information for a stage of processing.

The operation described is run on each tile of the image; *numTileX* and *numTileY* hold the number of tiles for this particular set of output images and input images.

*flags* holds generic information for the stage:

**Table 1 Stage Flags**

EDGE_POST_BREAKPOINT	Execute a breakpoint instruction before calling the tile function. This flag can be used selectively for debugging purposes.
EDGE_POST_WRITE_RSX_LABEL	If this flag is set, Edge Post will transfer the value specified in <i>rsxLabelValue</i> to the address specified in <i>rsxLabelAddress</i> when the operation is completed.

*rsxLabelAddress* is an optional pointer to an RSX™ label that will be updated with a value specified in *rsxLabelValue* once this particular operation is terminated and all output data has been transferred to memory. This field can be used to signal RSX™ that results are now available for consumption.

*effectCodeEa* is an optional effective address of SPU code that will be uploaded into SPU local store and executed for each tile of the image, *effectCodeSize* is the size of the effect code in local store, and *effectCodeDmaSize* is the size of the DMA required to transfer the code.

These fields are optional; if you do not need to load any code or will do your own code loading, set the address to zero. However, if you set the address to zero but the size is non-zero, Edge Post will still allocate the memory but it will not do any DMA (it assumes that you will do it). So if you do not need any of this functionality, set the size to zero too.

Each effect can have an assigned parameter area that is transferred with the code. The pointer to this area is stored in *stageParametersEa* and its size is stored in *stageParametersSize*.

*sources* is an array of up to four image descriptors for the stage. Refer to [EdgePostSourceTile](#) for more information.

---

# EdgePostWorkload

---

Structure used to hold a currently executing effect chain.

## Definition

---

```
#include <edgepost.h>
struct EdgePostWorkload
{
    /* omitted */
} __attribute__((__aligned__(128)));
```

## Description

---

This structure is always 128-byte aligned and 16 bytes in size. It is accessed from SPU's using atomic operations and used to synchronize many SPU's working on different tiles of the same effect chain.

---

# EdgePostTileInfo

---

Structure passed as parameter to the user-supplied tile callback.

## Definition

---

```
#include <edgepost_framework_spu.h>
struct EdgePostTileInfo
{
    EdgePostProcessStage* stage;
    uint32_t tile_x;
    uint32_t tile_y;
    uint8_t* tiles[EDGE_POST_MAX_TILES];
    void* parameters;
};
```

## Members

---

<i>stage</i>	Pointer to current processing stage descriptor
<i>tile_x</i>	Current tile X coordinate
<i>tile_y</i>	Current tile Y coordinate
<i>tiles</i>	Array of pointers to tile data
<i>parameters</i>	Pointer to stage specific parameters

## Description

---

This structure is passed as a parameter to the user-supplied tile function.

# EdgePostSpuConfig

Structure used to initialize the Edge Post tiling framework running on SPU.

## Definition

```
#include <edgepost_framework_spu.h>
struct EdgePostSpuConfig
{
    void* heapStart;
    uint32_t heapSize;
    uint16_t controlDmaTag;
    uint16_t inputDmaTag;
    uint16_t outputDmaTag;
    EdgePostPollCallback pollCallback;
    EdgePostStageStartCallback stageStartCallback;
    EdgePostStageEndCallback stageEndCallback;
} __attribute__((__aligned__(16)));
```

## Members

<i>heapStart</i>	Start of local store area assigned to Edge Post
<i>heapSize</i>	Size of local store area assigned to Edge Post
<i>controlDmaTag</i>	DMA tag index used for generic transfers
<i>inputDmaTag</i>	DMA tag index used for DMA gets
<i>outputDmaTag</i>	DMA tag index used for DMA puts
<i>pollCallback</i>	Callback used to check if yielding is needed
<i>stageStartCallback</i>	Callback called when a new effect stage starts
<i>stageEndCallback</i>	Callback called when current effect stage ends

## Description

Each SPU running Edge Post needs to initialize all fields of this structure and pass it as a parameter to [edgePostSetSpuConfig](#) before any additional operations.

*heapStart* and *heapSize* need to be initialized to the start and the size of the area in SPU local store that will be used internally by Edge Post.

*controlDmaTag*, *inputDmaTag*, and *outputDmaTag* are DMA tags that you are willing to hand over to Edge Post for its own internal transfers.

*pollCallback* is a pointer to a function of type [EdgePostPollCallback](#) that will be called at different times during tile processing to let the user poll for higher priority workloads to be run on the current SPU.

*stageStartCallback* is a pointer to a function of type `EdgePostStageStartCallback` that will be called every time a new stage in the effect chain is about to start.

*stageEndCallback* is a pointer to a function of type `EdgePostStageEndCallback` that will be called every time the current processed stage ends.

## Notes

Always make sure that *heapSize* is large enough to hold your tiles. Keep this value in sync with the one used on PPU when calculating tile sizes.

---

# EdgePostImage

---

Structure used by PPU helper functions to describe an input or output image memory layout.

## Definition

---

```
#include <edgepost_ppu.h>
struct EdgePostImage
{
    EdgePostTileDir m_dir;
    uint32_t m_ea;
    uint32_t m_pitch;
    uint16_t m_width;
    uint16_t m_height;
    uint8_t m_pixelSize;
    uint8_t m_borders[2];
    uint8_t m_multiplier;
};
```

## Members

---

<i>m_dir</i>	Tells whether the image is an input or output
<i>m_ea</i>	Effective address of the buffer holding the image
<i>m_pitch</i>	Pitch of the image
<i>m_width</i>	Width of the image in pixels
<i>m_height</i>	Height of the image in pixels
<i>m_pixelSize</i>	Size in bytes for one pixel
<i>m_borders</i>	Required borders in pixel for each tile
<i>m_multiplier</i>	Space multiplier used when calculating local store occupancy

## Description

---

This PPU structure is used as input to a couple of PPU functions provided to help select correct tile sizes and/or to set up [EdgePostProcessStage](#) structures.

# EdgePostMlaaContext

Structure used by PPU helper functions to store the state of an MLAA (Morphological Anti-aliasing) system

## Definition

```
#include <edgepost_mlaa_handler_ppu.h>
struct EdgePostMlaaContext
{
    uint32_t spuCount;
    CellSpurs* spurs;
    CellSpursTaskset2* taskSet;
    CellSpursTaskId* taskIds;
    CellSpursAttributes2* taskAttributes;
    CellSpursTaskArgument* taskArguments;
    EdgePostMlaaTaskParameters* taskParameters;
    CellSpursTaskSaveConfig* saveConfigs;
    CellSpursBarrier* barrier;
    int32_t* directionLock;
    uint32_t rsxLable;
    volatile uint32_t* rsxLabelAddress;
    uint32_t tasksReady;
}
```

## Members

<i>spuCount</i>	Number of logical SPUs to be used. Corresponds to number of tasks.
<i>spurs</i>	Pointer to the SPURS instance used.
<i>taskSet</i>	Pointer to the SPURS taskset used for this MLAA instance.
<i>taskIds</i>	Pointer to an array of SPURS task IDs of MLAA tasks. This field is only valid after calling function <a href="#">edgePostMlaaInitializeContext()</a> and is mainly for internal use.
<i>taskAttributes</i>	Pointer to an array of <i>spuCount</i> task attributes.
<i>taskArguments</i>	Pointer to an array of <a href="#">EdgePostMlaaTaskParameters()</a> . This field is only valid after calling function <a href="#">edgePostMlaaInitializeContext()</a> and is mainly for internal use.
<i>taskParameters</i>	Pointer to an array of <i>spuCount</i> task parameters.
<i>saveConfigs</i>	Pointer to an array of <i>spuCount</i> task save configurations.
<i>barrier</i>	Pointer to a SPURS barrier used to synchronize SPUs between MLAA and the transpose passes.
<i>directionLock</i>	Pointer to 128-byte aligned memory location.
<i>rsxLabel</i>	Label used to release RSX™ after the process has finished.
<i>rsxLabelAddress</i>	Address of the RSX™ label.
<i>tasksReady</i>	This value will be set by the SPUs when the operation is finished.

## Description

This PPU structure is used by the PPU side of MLAA to store instance information. It is recommended that you use [edgePostMlaaInitializeContext\(\)](#) to fill this structure.



# EdgePostMlaaTaskParameters

Structure used to facilitate parameter passing to the MLAA SPU task.

## Definition

```
#include <edgepost_mlaa.h>
struct EdgePostMlaaTaskParameters
{
    uint32_t rsxLabelValue;
    uint32_t rsxLabelAddress;
    uint32_t taskCounterAddress;
    uint32_t imageAddress;
    uint32_t destAddress;
    uint32_t barrierAddress;
    uint32_t directionLockAddress;
    uint16_t imageWidth;
    uint16_t imageHeight;
    uint16_t imagePitch;
    uint8_t mode;
    uint8_t spuId : 4;
    uint8_t spuCount : 4;
    uint16_t parameter0;
    uint16_t parameter1;
    uint8_t reserved[24];
};
```

## Members

<i>rsxLabelValue</i>	Value to be written to the RSX™ label once processing has finished.
<i>rsxLabelAddress</i>	Effective address of RSX™ label, or NULL if no writing is required.
<i>taskCounterAddress</i>	Effective address of a counter that will be incremented every time an SPU finishes. Can be set to NULL.
<i>imageAddress</i>	Effective address of the source image buffer. Must be 16-byte aligned, with 128-byte alignment strongly recommended.
<i>destAddress</i>	Effective address of the destination image buffer. Must be 16-byte aligned, with 128-byte alignment strongly recommended. Can be identical to <i>imageAddress</i> .
<i>barrierAddress</i>	Effective address of a SPURS barrier set up for <i>spuCount</i> SPU.
<i>directionLockAddress</i>	Effective address of a 128-byte aligned 4-byte XDR memory location initialized to 0.
<i>imageWidth</i>	Number of pixels per scanline that should be processed.
<i>imageHeight</i>	Height of the image, not including padding.
<i>imagePitch</i>	Horizontal pitch of the image buffer, in bytes.
<i>mode</i>	MLAA mode. See Description.
<i>spuId</i>	Number of this SPU within the task group.
<i>spuCount</i>	Size of task group.
<i>parameter0</i>	Edge detection base value. See Description.
<i>parameter1</i>	Edge detection scale. See Description.
<i>reserved</i>	Reserved for internal use.

## Description

Each MLAA task receives a copy of this structure. It will usually be filled by [edgePostMlaaPrepareWithRelativeThreshold\(\)](#).

The task supports two ways of signaling completion: an RSX™ label and an XDR counter. If *rsxLabelAddress* is not equal to 0, the SPU will write the *rsxLabelValue* after synchronizing

with all SPUs. Similarly, if *taskCounterAddress* is not 0, the SPU will increment the value after finishing and synchronizing. Note that these can be set per SPU and that writing happens after all SPUs are finished, so in general only one SPU will have one or two of these addresses set.

The size of the buffer pointed to by *imageAddress* is assumed to be *imagePitch\*imageHeight*; moreover, the buffer pointed to by *destAddress* must have a height that is divisible by 128. Thus, for a 720p buffer, space for 1280\*768 pixels is required. There is no performance penalty for using the same buffer for both input and output.

Because the buffer pointed to by *directionLockAddress* is repeatedly used during the transpose passes, it is recommended that it lay in its own cache-line to prevent false sharing.

The parameter *parameter0* defines the lower bound for pixel thresholds, and *parameter1* is a 0.8 fixed-point encoding of a scale. For each pixel, the RGB channel with the highest numerical value is multiplied by this scale. The maximum of this value and the lower bound from *parameter1* defines the threshold used for this pixel.

During edge detection, the criterion for finding an edge is that the absolute difference in any color channel between two pixels is greater than the minimum of the thresholds of these two pixels.

MLAA can operate in several different ways, which can be selected using the *mode* parameter. Flags can be safely combined, even if only certain combinations are useful. The complete list of supported flags is as follows:

EDGE_POST_MLAA_MODE_ENABLED	If this flag is not set, the task will perform a simple copy from <i>imageAddress</i> to <i>destAddress</i> .
EDGE_POST_MLAA_MODE_SHOW_EDGES	Mark horizontal edges red and vertical edges green.
EDGE_POST_MLAA_MODE_SHUTDOWN	Shut down tasks. If this mode is set, no MLAA operation will be performed and all parameters except for mode can be undefined.
EDGE_POST_MLAA_MODE_SINGLE_SPU_TRANSPOSE	Perform the transpose operation on one SPU instead of two. This reduces XDR peak-pressure and minimizes total SPU usage, at the cost of additional latency.
EDGE_POST_MODE_TRANSPOSE_64	Change the transpose block size from 128*128 to 64*64. This changes the restriction for width and padded height to multiples of 64, instead of 128, at the cost of performance.

## Notes

An absolute-difference thresholding can be performed by setting *parameter1* to 0 and putting the desired threshold into *parameter0*. This gives the thresholding function used in the original paper.

---

# EdgePostMlaaMemoryLayout

---

Working memory for the SPU MLAA functions.

## Definition

---

```
#include <edgepost_mlaa_memory_layout.h>
struct EdgePostMlaaMemoryLayout
{
    /* omitted */
};
```

## Description

---

When calling the MLAA functions on the SPU, all working memory is passed in the form of this structure.

Between calls to the MLAA functions, the contents do not need to be preserved, and preserving them has no performance benefits.

## Notes

---

If the application uses the provided MLAA SPURS task, there is no need to call the SPU MLAA functions directly and thus no need to use this structure.

The size of the structure is close to that of local store.

---

# EdgePostInplaceTransposeMemoryLayout

---

Working memory for the SPU in-place pseudo-transpose function.

## Definition

---

```
#include <edgepost_inplace_transpose.h>
struct EdgePostInplaceTransposeMemoryLayout
{
    /* omitted */
};
```

## Description

---

When calling the in-place transpose functions on the SPU, all working memory is passed in the form of this structure.

Between calls to the in-place transpose functions, the contents do not need to be preserved, and preserving them has no performance benefits.

## Notes

---

The size of the structure is close to that of local store.

---

# EdgePostInplaceTransposePostOpFunction

---

Function type used when performing a secondary operation during transpose.

## Definition

---

```
#include <edgepost_inplace_transpose.h>
typedef void (*EdgePostInplaceTransposePostOpFunction)
    (void* param,
     void* data,
     size_t dataSize
    );
```

## Members

---

<i>param</i>	A user-defined parameter passed to the function
<i>data</i>	Pointer to the transposed data buffer
<i>dataSize</i>	Size of of the buffer pointed to by <i>data</i> , in bytes

## Description

---

Functions of this type can be passed to [edgePostTransposeInPlace\(\)](#).

## Notes

---

The intention is to perform simple operations at times when the SPU would otherwise stall while waiting for a DMA request to finish. As a guideline, approximately 8 cycles per quadword in *data* can be hidden in the transpose operation.

## See Also

---

[edgePostTransposeInPlace](#)

# SPU Functions

---

# edgePostSetSpuConfig

---

Configures and initializes the SPU tiling framework.

## Definition

---

```
#include <edgepost_framework_spu.h>
void edgePostSetSpuConfig(
    const EdgePostSpuConfig* config
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Not multithread safe.

## Arguments

---

<i>config</i>	Tiling framework configuration
---------------	--------------------------------

## Return Values

---

None.

## Examples

---

```
// allocate heap area
const int kHeapSize = 1024 * 200;
void* pHeap = memalign( 128, kHeapSize );
EDGE_ASSERT( pHeap && "no memory" );

// setup edgePost configuration
EdgePostSpuConfig config;
config.heapStart = pHeap;
config.heapSize = kHeapSize;
config.controlDmaTag = 0;
config.inputDmaTag = 1;
config.outputDmaTag = 2;
config.stageStartCallback = _StageStartCallback;
config.stageEndCallback = _StageEndCallback;
config.pollCallback = _Poll;

// set configuration
edgePostSetSpuConfig( &config );
```

## Notes

---

Always call this function before starting any Edge Post workload on SPU.

## See Also

---

[EdgePostSpuConfig](#)

---

# edgePostRunWorkload

---

Starts or joins execution of an Edge Post Workload.

## Definition

```
#include <edgepost_framework_spu.h>
int edgePostRunWorkload(
    uint32_t workloadEa
)
```

## Calling Conditions

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Not multithread safe.

## Arguments

*workloadEa*      Effective address of an initialized [EdgePostWorkload](#)

## Return Values

Returns one of the following codes:

Macro	Value	Description
EDGEPOST_WORKLOAD_ENDED	0	Execution of current workload ended.
EDGEPOST_WORKLOAD_PREEMPTED	1	Another task, job, or workload with higher priority needs to be executed.
EDGEPOST_WORKLOAD_IDLE	2	Workload is not finished but the current SPU needs to wait for another SPU to finish the current stage in the effect chain.

## Description

This function starts execution of an Edge Post workload. If the workload is being worked on by other SPUs, the SPU calling this function will help out execution, sharing work with other SPUs.

This function can return for one of the following reasons:

- An effect chain is finished.
- Another Task/Job with higher priority wants to take control of the SPU.
- Another SPU is terminating the last tile of the current effect stage. Any other SPU cannot proceed to the next stage until the current stage has been finalized.

It is acceptable to call this function in a tight loop, checking for the `EDGEPOST_WORKLOAD_ENDED` condition to be met to exit the loop.

Alternatively, the SPU has the opportunity to do some unrelated work when the function returns `EDGEPOST_WORKLOAD_IDLE` before calling the same function again; or to yield the SPU to another workload with higher priority.



**Examples**

---

```
while (1)
{
    // call main execution function
    int ret = edgePostRunWorkload( workloadEa );

    if ( ret == EDGEPOST_WORKLOAD_PREEMPTED)
    {
        // Yield here if you want to share this SPU with others
        // for example call cellSpursYield()
    }

    if ( ret == EDGEPOST_WORKLOAD_ENDED)
    {
        // workload is finished, exit the task
        break;
    }

    // workload is not finished but this particular SPU is currently idle
    // because it needs to wait for other SPUs to finish last tiles of
    // current stage
}
```

---

# edgePostMlaaPass

---

Performs a single horizontal or vertical MLAA pass.

## Definition

---

```
#include <edgepost_mlaa.h>
void edgePostMlaaPass
(
    EdgePostMlaaMemoryLayout* layout,
    const EdgePostMlaaTaskParameters* parameter,
    uint32_t pass,
    uint32_t first,
    uint32_t baseDmaTag
)
```

## Calling Conditions

---

Not multithread safe.

## Arguments

---

<i>layout</i>	Pointer to working memory
<i>parameter</i>	Pointer to the Task parameters
<i>pass</i>	0 for a horizontal and 1 for a vertical pass
<i>first</i>	1 for the first pass, 0 for all others
<i>baseDmaTag</i>	Lowest DMA tag used

## Return Values

---

None

## Description

---

This function performs either a horizontal or vertical pass, based on the *pass* parameter. Vertical passes are expecting the input data to be in pseudo-transposed form, which can be achieved by running [edgePostInplaceTransposePass\(\)](#) on the source buffer.

## Notes

---

For performance reasons, it is advisable to run the vertical pass first, especially if more than two SPUs are used for the operation.

The function will use 18 DMA tags beginning with *baseDmaTag*. These may safely overlap with the tags used by [edgePostInplaceTransposePass\(\)](#).

## See Also

---

[edgePostInplaceTransposePass](#)

---

# edgePostInplaceTransposePass

---

Performs an in-place pseudo transpose pass for use with MLAA.

## Definition

---

```
#include <edgepost_mlaa.h>
void edgePostInplaceTransposePass
(
    EdgePostInplaceTransposeMemoryLayout* layout,
    const EdgePostMlaaTaskParameters* parameter,
    bool first,
    uint32_t baseDmaTag
)
```

## Calling Conditions

---

Not multithread safe.

## Arguments

---

<i>layout</i>	Pointer to working memory
<i>parameter</i>	Pointer to the Task parameters
<i>first</i>	1 for the first pass, 0 for all others
<i>baseDmaTag</i>	Lowest DMA tag used

## Return Values

---

None

## Description

---

This function performs a pseudo-transpose from the buffer pointed to by the *imageAddress* member of *parameter* to the buffer pointed to by the *destAddress* member.

## Notes

---

If this is the first pass in an MLAA operation, it is important to set the *imageAddress* to *destAddress* for subsequent passes.

Use this function rather than calling [edgePostTransposeInPlace\(\)](#) directly, because it will perform the transpose on two SPU's if possible, which is the ideal number. It can be called by all tasks, even the ones that do not contribute to the operation.

The function will use three DMA tags beginning with *baseDmaTag*. These may safely overlap with the tags used by [edgePostMlaaPass\(\)](#).

## See Also

---

[edgePostMlaaPass](#), [edgePostTransposeInPlace](#)

# edgePostTransposeInPlace

Performs a 128x128 pixel pseudo-transpose on a 32-bpp image.

## Definition

```
#include <edgepost_inplace_transpose.h>

void edgePostTransposeInPlace
(
    EdgePostInplaceTransposeMemoryLayout* layout,
    uint32_t sourceEa,
    uint32_t destEa,
    uint32_t imagePitch,
    uint32_t imageHeight,
    uint32_t spuId,
    uint32_t lockAddress,
    EdgePostInplaceTransposePostOpFunction postOpFunction,
    void* postOpParameter,
    uint32_t baseDmaTag
)
```

## Calling Conditions

Not multithread safe.

## Arguments

<i>layout</i>	Pointer to working memory.
<i>sourceEa</i>	Effective address of source buffer.
<i>destEa</i>	Effective address of destination buffer.
<i>imagePitch</i>	Pitch of buffer. Must be a multiple of 128.
<i>imageHeight</i>	Height of buffer. Must be a multiple of 128.
<i>spuId</i>	ID of the SPU, either 0 or 1.
<i>lockAddress</i>	Effective address to a 128-byte aligned buffer in main memory. See Notes.
<i>postOpFunction</i>	Pointer to a function called after the transpose operation, or NULL.
<i>postOpParameter</i>	User-defined parameter to be passed to <i>postOpFunction</i> , or NULL.
<i>baseDmaTag</i>	Lowest DMA tag used

## Return Values

None

## Description

This function is used to compute a pseudo-inverse of an image, by transposing pixels within 128x128 pixel blocks. An algorithm can then reconstruct vertical pixel-columns by piecing together these 128 pixel segments. Because the actual transposing only consumes approximately half the SPU time, it is possible to perform an operation on the transposed buffer while waiting for the DMA transfers.

The function will use three DMA tags beginning with *baseDmaTag*.

## Notes

The operation is split into two parts, which can be performed sequentially or in parallel.

Transposing a 720p buffer with two SPUs takes approximately 400 µs per SPU.

The *lockAddress* is used for a synchronization primitive inside the function. Because this primitive reads and writes to the memory using atomic operations, it is recommended that you not use the cache line for other purposes. Furthermore, the value pointed to by *lockAddress* must be 0 when the function is called, and it will leave the value at 0 when it exits.

### **See Also**

---

`edgePostTransposePass`

# SPU Processing Functions

---

# edgePostDownsample8

---

Bilinear down-sampling.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostDownsample8
(
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>stride</i>	Stride in bytes for one row of input
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function perform a 2x2 bilinear down-sampling of two rows of an image. The result is one down-sampled row.

Both *input* and *output* images are formatted as argb 32 bit.

*count* specifies the number of pixels per row after down-sampling.

The number of input pixel per row must be a multiple of 8.

---

# edgePostDownsample16

---

Bilinear down-sampling.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostDownsample16
(
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>stride</i>	Stride in bytes for one row of input
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function perform a 2x2 bilinear down-sampling of two rows of an image. The result is one down-sampled row.

Both *input* and *output* images are formatted as argb 16 bits per channel; channels are unsigned integer.

*count* specifies the number of pixels per row after down-sampling.

The number of input pixel per row must be a multiple of 8.



---

# edgePostDownsampleF

---

Bilinear down-sampling.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostDownsampleF
(
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>stride</i>	Stride in bytes for one row of input
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function perform a 2x2 bilinear down-sampling of two rows of an image. The result is one down-sampled row.

Both *input* and *output* images are single-channel floating-point images.

*count* specifies the number of pixels per row after down-sampling.

The number of input pixels per row must be a multiple of 8.

---

# edgePostNearestDownsample

---

Nearest down-sampling.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostNearestDownsample
(
    void* output,
    const void* input,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function perform a 2x2 down-sampling of two rows of an image with pixel size equal to 4.

The sample chosen is always the top-left of the 2x2 quad.

The number of input pixels per row must be a multiple of 8.

---

# edgePostDownsampleFloatMin

---

Down-samples a floating-point image by choosing the minimum value from the samples.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostDownsampleFloatMin
(
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>stride</i>	Stride in bytes for one row of input
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function performs a 2x2 down-sampling of two rows of a one-channel floating-point image into one row. The candidate sample chosen is the minimum value between the four samples.

This function can be used, for example, to down-sample a depth image and always choose the nearest sample to the camera.

---

# edgePostDownsampleFloatMax

---

Down-samples a floating-point image by choosing the maximum value from the samples.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostDownsampleFloatMax
(
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>stride</i>	Stride in bytes for one row of input
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function performs a 2x2 down-sampling of two rows of a one channel floating-point image into one row. The candidate sample chosen is the maximum value between the four samples.

This function can be used, for example, to down-sample a depth image and always choose the farthest sample from the camera.

---

# edgePostUpsample8

---

Bilinear up-sampling.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostUpsample8
(
    void* output,
    const void* input,
    uint32_t istride,
    uint32_t ostride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>istride</i>	Stride in bytes for one row of input
<i>ostride</i>	Stride in bytes for one row of output
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function performs bilinear up-sampling.

*input* is one row of an image; *output* will be two rows with double the width.

For each invocation, the function needs to access previous and next input rows; so make sure to have enough borders along your input tile edges: a minimum of one row of border is required for the top/bottom, and a minimum of one pixel on the left/right edge of the tile.

*count* is the number of pixels in an output row; it must be a multiple of 8 pixels.

This function works on four channel images, where each channel is 8 bits.

---

# edgePostUpsample16

---

Bilinear up-sampling.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostUpsample16
(
    void* output,
    const void* input,
    uint32_t istride,
    uint32_t ostride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>istride</i>	Stride in bytes for one row of input
<i>ostride</i>	Stride in bytes for one row of output
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function performs bilinear up-sampling.

*input* is one row of an image; *output* will be two rows with double the width.

For each invocation, the function needs to access previous and next input rows; so make sure to have enough borders along your input tile edges: a minimum of one row of border is required for the top/bottom, and a minimum of one pixel on the left/right edge of the tile.

*count* is the number of pixels in an output row; it must be a multiple of 8 pixels.

This function works on four channel images, where each channel is 16 bits.

---

# edgePostUpsampleF

---

Bilinear up-sampling.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostUpsampleF
(
    void* output,
    const void* input,
    uint32_t istride,
    uint32_t ostride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>istride</i>	Stride in bytes for one row of input
<i>ostride</i>	Stride in bytes for one row of output
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function performs bilinear up-sampling.

*input* is one row of an image; *output* will be two rows with double the width.

For each invocation, the function needs to access previous and next input rows; so make sure to have enough borders along your input tile edges: a minimum of one row of border is required for the top/bottom, and a minimum of one pixel on the left/right edge of the tile.

*count* is the number of pixels in an output row; it must be a multiple of 8 pixels.

This function works on single-channel images, where the channel is in floating-point format.

---

# edgePostGauss7x1\_8

---

Horizontal, 7-pixel wide Gaussian blur.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostGauss7x1_8
(
    void* output,
    const void* input,
    uint32_t count,
    vec_float4 weights
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels to process
<i>weights</i>	Packed Gaussian blur weights

## Return Values

---

None.

## Description

---

This function performs a horizontal, 7-pixel wide Gaussian blur on an input row of pixels.

*weights* holds Gaussian weights to be used in the filter operation.

This function requires a left/right border of at least 3 pixels.

This function works on four channel images, where each channel is 8 bits.

```
result =
    weight.w x input[-3,0] +
    weight.z x input[-2,0] +
    weight.y x input[-1,0] +
    weight.x x input[0,0] +
    weight.y x input[1,0] +
    weight.z x input[2,0] +
    weight.w x input[3,0]
```



---

# edgePostGauss7x1F

---

Horizontal, 7-pixel wide Gaussian blur

## Definition

---

```
#include <edgepost_spu.h>
void edgePostGauss7x1F
(
    void* output,
    const void* input,
    uint32_t count,
    vec_float4 weights
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels to process
<i>weights</i>	Packed Gaussian blur weights

## Return Values

---

None.

## Description

---

This function performs a horizontal, 7-pixel wide Gaussian blur on an input row of pixels.

*weights* holds Gaussian weights to be used in the filter operation.

This function requires a left/right border of at least 3 pixels.

This function works on single-channel images, where the channel is in floating-point format.

```
result =
    weight.w x input[-3,0] +
    weight.z x input[-2,0] +
    weight.y x input[-1,0] +
    weight.x x input[0,0] +
    weight.y x input[1,0] +
    weight.z x input[2,0] +
    weight.w x input[3,0]
```

---

# edgePostGauss1x7\_8

---

Vertical, 7-pixel wide Gaussian blur.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostGauss1x7_8
(
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count,
    vec_float4 weights
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>stride</i>	Size in bytes for one input row
<i>count</i>	Number of pixels to process
<i>weights</i>	Packed Gaussian blur weights

## Return Values

---

None.

## Description

---

This function performs a vertical, 7-pixel wide Gaussian blur on an input row of pixels.

*weights* holds Gaussian weights to be used in the filter operation.

This function requires a top/bottom border of at least 3 rows.

This function works on four channel images, where each channel is 8 bit.

```
result =
    weight.w x input[0,-3] +
    weight.z x input[0,-2] +
    weight.y x input[0,-1] +
    weight.x x input[0,0] +
    weight.y x input[0,1] +
    weight.z x input[0,2] +
    weight.w x input[0,3]
```

---

# edgePostGauss1x7F

---

Vertical, 7-pixel wide Gaussian blur.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostGauss1x7F
(
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count,
    vec_float4 weights
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>stride</i>	Size in bytes for one input row
<i>count</i>	Number of pixels to process
<i>weights</i>	Packed Gaussian blur weights

## Return Values

---

None.

## Description

---

This function performs a vertical, 7-pixel wide Gaussian blur on an input row of pixels.

*weights* holds Gaussian weights to be used in the filter operation.

This function requires a top/bottom border of at least 3 rows.

This function works on single-channel images, where channel is in floating-point format.

```
result =
    weight.w x input[0,-3] +
    weight.z x input[0,-2] +
    weight.y x input[0,-1] +
    weight.x x input[0,0] +
    weight.y x input[0,1] +
    weight.z x input[0,2] +
    weight.w x input[0,3]
```

# edgePostBloomCapture8

Captures bloom color from a source image.

## Definition

```
#include <edgepost_spu.h>
edgePostBloomCapture8
(
    void* output,
    const void* input,
    uint32_t count,
    vec_float4 exposureLevel,
    vec_float4 minLuminance,
    vec_float4 luminanceRangeRcp
)
```

## Calling Conditions

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels to process
<i>exposureLevel</i>	Luminance multiplier factor
<i>minLumimance</i>	Threshold luminance above which the pixel will contribute to bloom
<i>luminanceRangeRcp</i>	$1 / (maxLuminance - minLuminance)$

## Return Values

None.

## Description

This function generates a row's worth of bloom colors from an original row of pixels.

Both input and output are argb8 images.

Bloom value is calculated this way :

```
luminance = max( RgbToLuminance( source_color * exposureLevel ), minLuminance )
scale = clamp(( luminance - minLuminance ) * luminanceRangeRcp, 0.0, 1.0 )
bloom = ( scale / luminance ).xxx * source_color;
```

# edgePostBloomCaptureFX16

Captures bloom color from a source image.

## Definition

```
#include <edgepost_spu.h>
edgePostBloomCaptureFX16
(
    void* output,
    const void* input,
    uint32_t count,
    vec_float4 exposureLevel,
    vec_float4 minLuminance,
    vec_float4 luminanceRangeRcp
)
```

## Calling Conditions

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels to process
<i>exposureLevel</i>	Luminance multiplier factor
<i>minLumimance</i>	Threshold luminance above which the pixel will contribute to bloom
<i>luminanceRangeRcp</i>	$1 / (maxLuminance - minLuminance)$

## Return Values

None.

## Description

This function generates a row worth of bloom colors from an original row of pixels.

Both input and output are fx16 images.

Bloom value is calculated this way :

```
luminance = max( RgbToLuminance( source_color * exposureLevel ), minLuminance )
scale = clamp(( luminance - minLuminance ) * luminanceRangeRcp, 0.0, 1.0 )
bloom = ( scale / luminance ).xxx * source_color;
```

---

# edgePostTonemapFX16

---

Applies tone-mapping to an image.

## Definition

---

```
#include <edgepost_spu.h>
edgePostTonemapFX16
(
    void* output,
    const void* input,
    uint32_t count,
    float avgLuminance,
    float middleGray,
    float whitePoint
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels to process
<i>avgLuminance</i>	Average luminance level for the input buffer
<i>middleGray</i>	Middle gray value
<i>whitePoint</i>	White point value

## Return Values

---

None.

## Description

---

This function applies tone-mapping and gamma-correction to an fx16 input buffer. Writes out an argb8 buffer.

---

# edgePostAvgLuminanceFX16

---

Calculates average luminance of an input image.

## Definition

---

```
#include <edgepost_spu.h>
edgePostAvgLuminanceFX16
(
    void* input,
    uint32_t count,
    vec_float4* avgLuminancePtr,
    vec_float4* maxLuminancePtr,
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels to process
<i>avgLuminancePtr</i>	Pointer to calculated average luminance
<i>maxLuminancePtr</i>	Pointer to calculated maximum luminance

## Return Values

---

None.

## Description

---

This function calculates average and maximum luminance level for an input fx16 image.

---

# edgePostModulate8

---

Modulates two input images together.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostModulate8
(
    void* output,
    const void* input0,
    qword shuffle0,
    const void* input1,
    qword shuffle1,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to first image
<i>shuffle0</i>	Shuffle mask used on pixels loaded from the first image
<i>input1</i>	Pointer to second image
<i>shuffle1</i>	Shuffle mask used on pixels loaded from the second image
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function multiplies two rows of argb8 pixels together, such as:

```
result = shuffle0(input0) x shuffle1(input1)
```



---

# edgePostModulateFX16

---

Modulates two input images together.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostModulateFX16
(
    void* output,
    const void* input0,
    qword shuffle0,
    const void* input1,
    qword shuffle1,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to first image
<i>shuffle0</i>	Shuffle mask used on pixels loaded from the first image
<i>input1</i>	Pointer to second image
<i>shuffle1</i>	Shuffle mask used on pixels loaded from the second image
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function multiplies two rows of fx16 pixels together, such as:

```
result = shuffle0(input0) x shuffle1(input1)
```

---

# edgePostConstantModulate8

---

Modulates one image with a constant factor.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostConstantModulate8(
    void* output,
    const void* input0,
    qword shuffle0,
    qword constant,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to first image
<i>shuffle0</i>	Shuffle mask used on pixels loaded from the first image
<i>constant</i>	Second operand
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

This function multiplies argb8 pixel of one row by the supplied constant value, such as:

```
result = shuffle0(input0) x constant
```

---

# edgePostAddSat8

---

Adds two images together.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostAddSat8(
    void* output,
    const void* input0,
    const void* input1,
    vec_uint4 multiplier,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to first image
<i>input1</i>	Pointer to second image
<i>multiplier</i>	Integer multiplier
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

Adds two rows of argb8 pixels together:

*result* = *input0* + *input1* x *multiplier*

---

# edgePostAddSatFX16

---

Adds two images together.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostAddSatFX16(
    void* output,
    const void* input0,
    const void* input1,
    vec_uint4 multiplier,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to first image
<i>input1</i>	Pointer to second image
<i>multiplier</i>	Integer multiplier
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

Adds two rows of fx16 pixels together:

$$\text{result} = \text{input0} + \text{input1} \times \text{multiplier}$$

---

# edgePostBlend8

---

Blends two images together.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostBlend8(
    void* output,
    const void* input0,
    const void* input1,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to first image
<i>input1</i>	Pointer to second image
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

Blends two rows of argb8 pixels together:

$$\text{result} = \text{input0} \times (1 - \text{input1.alpha}) + \text{input1}$$

---

# edgePostBlendFX16

---

Blends two images together.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostBlendFX16(
    void* output,
    const void* input0,
    const void* input1,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to first image
<i>input1</i>	Pointer to second image
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

Blends two rows of fx16 pixels together:

$$\text{result} = \text{input0} \times (1 - \text{input1.alpha}) + \text{input1}$$

---

# edgePostPremultiplyFX16

---

Pre-multiplies by alpha channel.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostPremultiplyFX16(
    void* output,
    const void* input0,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to input image
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

Pre-multiplies input image by the alpha channel:

$\text{result} = \text{input0} \times \text{input0.alpha}$

---

# edgePostCombine

---

Combines pixels from two different images using a shuffle mask.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostCombine(
    void* output,
    const void* input0,
    const void* input1,
    qword shuffleMask,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input0</i>	Pointer to first image
<i>input1</i>	Pointer to second image
<i>shuffleMask</i>	Shuffle mask to be used in the combination
<i>count</i>	Number of pixels to process

## Return Values

---

None.

## Description

---

Combines two rows of pixels together, processing 16 bytes at a time :

```
result = spu_shuffle(input0, input1, shuffleMask)
```



---

# edgePostExtractDepth

---

Converts an RSX™ depth buffer to linear depth.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostExtractDepth(
    void* output,
    const void* input,
    uint32_t count,
    float near,
    float far
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input depth buffer in RSX™ format
<i>count</i>	Number of pixels
<i>near</i>	Location of the near plane
<i>far</i>	Location of the far plane

## Return Values

---

None.

## Description

---

This function can be used to convert depth buffers in RSX™ format (D24S8) into a single-channel floating-point image where the value 0 maps to the near plane and the value 1 maps to the far plane.

---

# edgePostFloatToGrayscale

---

Converts a floating-point single-channel image into a gray scale argb8 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFloatToGrayscale(
    void* output,
    const void* input,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

This function converts a one-channel floating-point image into a gray scale argb8 image. It is mainly supplied for debugging purposes. Conversion can happen in-place, so the input pointer can be equal to the output pointer.

---

# edgePostArgb8ToFloats

---

Converts from argb8 to a floating-point four-channel image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostArgb8ToFloats (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts an input image in the format argb8 with 4 bytes per pixel into a four-channel floating-point image with 16 bytes per pixels.

Note that this function can convert data in place, so *input* and *output* can point to the same area of memory. Make sure that you have enough space allocated for the output image.

This function combined with the *multiplier* modifier for an input tile lets you convert your tiles in place to a more accessible format in terms of SPU code (each pixel being 16 bytes in size).

---

# edgePostFloatsToArgb8

---

Converts from a floating-point four-channel image to an argb8 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFloatsToArgb8(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a four-channel floating-point image into an argb8 image.

Note that this function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFX16ToFloats

---

Converts from fixed-point 5:11 to a floating-point four-channel image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFX16ToFloats (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts an input image in fixed-point 5:11 into a four-channel floating-point image with 16 bytes per pixel. This function can convert data in place, so *input* and *output* can point to the same area of memory. Make sure that you have enough space allocated for the output image.

---

# edgePostFX16ToArgb8

---

Converts from fixed-point 5:11 to an argb8 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFX16ToArgb8(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts an input image in fixed-point 5:11 into an argb8 image.

Note that this function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFP16LoToFloats

---

Converts low FP16 of each 32-bit word into a single-channel floating image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFP16LoToFloats (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Input pixel format should be 4 bytes in size. The high 16 bits of the data are discarded. The low 16 bits of the data are converted from FP16 to a 32-bit floating point.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFP16HiToFloats

---

Converts high FP16 of each 32-bit word into a single-channel floating image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFP16HiToFloats (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Input pixel format should be 4 bytes in size. The low 16 bits of the data are discarded. The high 16 bits of the data are converted from FP16 to a 32-bit floating point.

This function can convert data in place, so *input* and *output* can point to the same area of memory.



---

# edgePostFP16ToFloats

---

Converts a four-channel FP16 image into an FP32 image

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFP16ToFloats (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a four-channel FP16 image into an FP32 image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFloatsToFP16

---

Converts a four-channel FP32 image into an FP16 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFloatsToFP16(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a four-channel FP32 image into an FP16 image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFX16ToFP16

---

Converts a four-channel fx16 (5:11) image into an FP16 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFX16ToFP16(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a four-channel fx16 image into an FP16 image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostLogLuvToFloats

---

Converts a 4-byte Log(L)uv image into a four-channel floating-point image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostLogLuvToFloats (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a 4-byte Log(L)uv image into a four-channel floating-point image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostLogLuvToFX16

---

Converts a 4-byte Log(L)uv image into a four-channel fixed-point 5:11 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostLogLuvToFX16 (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a 4-byte Log(L)uv image into a four-channel fixed-point 5:11 image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostLogLuvToArgb

---

Converts a 4-byte Log(L)uv image into an argb8 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostLogLuvToArgb (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a 4-byte Log(L)uv image into an argb8 image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFloatsToLogLuv

---

Converts a 4-channel floating-point image into a Log(L)uv image

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFloatsToLogLuv(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a 4-channel floating-point image into a Log(L)uv image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFX16ToLogLuv

---

Converts a 4-channel fixed-point 5:11 image into a Log(L)uv image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFX16ToLogLuv(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a 4-channel fixed-point 5:11 to a Log(L)uv image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.



---

# edgePostFloatsToLuv

---

Converts a 4-channel floating-point image into a Luv image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFloatsToLuv(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a 4-channel floating-point image into a Luv image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFX16ToLuv

---

Converts an FX16 image into a Luv image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFX16ToLuv(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts an FX16 image into a Luv image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostLuvToFloats

---

Converts a Luv image into a four-channel floating-point image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostLuvToFloats(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a Luv image into a four-channel floating-point image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostLuvToFX16

---

Converts a Luv image into an FX16 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostLuvToFX16(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a Luv image into an FX16 image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostLuvToArgb

---

Converts a Luv image into an argb8 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostLuvToArgb(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a Luv image into an argb8 image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFP16LuvToFloats

---

Converts an FP16Luv image into a four-channel floating-point image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFP16LuvToFloats (
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts an FP16Luv image into a four-channel floating-point image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFP16LuvToFX16

---

Converts an FP16Luv image into an fx16 image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFP16LuvToFX16(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts an FP16Luv image into an fx16 image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

---

# edgePostFloatsToFP16Luv

---

Converts a four-channel floating-point image into an FP16Luv image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFloatsToFP16Luv(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts a four-channel floating-point image into an FP16Luv image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.



---

# edgePostFX16ToFP16Luv

---

Converts an fx16 image into an FP16Luv image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostFX16ToFP16Luv(
    void* output,
    const void* input,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input pixels
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

Converts an fx16 image into an FP16Luv image.

This function can convert data in place, so *input* and *output* can point to the same area of memory.

# edgePostDof

Calculates depth of field on an input image.

## Definition

```
#include <edgepost_spu.h>
void edgePostDof (
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count,
    const vec_float4* tapOffsetTable
)
```

## Calling Conditions

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input image
<i>stride</i>	Stride in pixel for the input image
<i>count</i>	Number of pixels
<i>tapOffsetTable</i>	A table of 16 <code>vec_float4</code> with tap offsets for the DOF filter

## Return Values

None.

## Description

This function calculates depth of field on an input image.

*input* must be a four-channel floating-point image where the alpha channel represents fuzziness. You can use `edgePostDofPremultiply` to convert an image to the correct format.

*tapOffsetTable* points to a table of 16 offsets defining the surrounding distribution of the samples taken when applying the blur kernel. For each entry, the first float is the X offset, the second float is the Y offset, and remaining values are not used. Offsets in this table also define the maximum radius of the blur, which also gives you an upper bound on the size of the required border for the input tile.

A default offset table is provided by the global variable:

```
extern const vec_float4 g_defaultDofTaps[];
```

If using this table, the borders of the tile need to be set at least to 8 for left/right and 7 for up/down.

The output of this function is a row of `argb8` pixels. Alpha is primed with a fuzziness value; this value can be used as a blending factor with the original unblurred image.

# edgePostDof\_FX16

Calculates depth of field on an input image.

## Definition

```
#include <edgepost_spu.h>
void edgePostDof_FX16(
    void* output,
    const void* input,
    uint32_t stride,
    uint32_t count,
    const vec_float4* tapOffsetTable
)
```

## Calling Conditions

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input image
<i>stride</i>	Stride in pixel for the input image
<i>count</i>	Number of pixels
<i>tapOffsetTable</i>	A table of 16 <code>vec_float4</code> with tap offsets for the DOF filter

## Return Values

None.

## Description

This function calculates depth of field on an input image.

*input* must be a four-channel floating-point image where the alpha channel represents fuzziness. You can use `edgePostDofPremultiply` to convert an image to the correct format.

*tapOffsetTable* points to a table of 16 offsets defining the surrounding distribution of the samples taken when applying the blur kernel. For each entry, the first float is the X offset, the second float is the Y offset, and remaining values are not used. Offsets in this table also define the maximum radius of the blur, which also gives you an upper bound on the size of the required border for the input tile.

A default offset table is provided by the global variable :

```
extern const vec_float4 g_defaultDofTaps[];
```

If using this table, the borders of the tile need to be set at least to 8 for left/right and 7 for up/down.

The output of this function is a row of `fx16` pixels. Alpha is primed with a fuzziness value; this value can be used as a blending factor with the original unblurred image.

---

# edgePostInitializeDofInputBuffer

---

Prepares an input image for subsequent call to [edgePostDof\(\)](#).

## Definition

---

```
#include <edgepost_spu.h>
void edgePostInitializeDofInputBuffer(
    void* output,
    const void* colors,
    const void* fuzziness,
    uint32_t count
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>colors</i>	Pointer to input color image
<i>fuzziness</i>	Pointer to input fuzziness image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

This helper function will take as input an argb8 color image and a “fuzziness” image and combine them into a single four-channel floating-point image suited to be input to `edgePostDof`.

The input fuzziness image must have been generated using [edgePostExtractFarFuzziness\(\)](#).

The operation of this function is as follows: the color image is converted to floating points, and the alpha of the result is primed using the input fuzziness.

---

# edgePostExtractNearFuzziness

---

Calculates near fuzziness for depth of field.

## Definition

---

```
#include <edgepost_framework_spu.h>
void edgePostExtractNearFuzziness(
    void* output,
    const void* input,
    uint32_t count,
    float nearFuzzy,
    float nearSharp,
    float maxFuzziness
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input image with depth information
<i>count</i>	Number of pixels
<i>nearFuzzy</i>	Normalized depth value; every pixel with depth smaller will be out of focus
<i>nearSharp</i>	Normalized depth value; every pixel with depth greater will be in focus
<i>maxfuzziness</i>	Maximum fuzziness value

## Return Values

---

None.

## Description

---

This function will generate a near fuzziness buffer from a depth buffer.

Both input and output images are one-channel floating-point images; input must contain linearized depth values.

Every pixel with depth less than *nearFuzzy* will have fuzziness of 1 (out of focus), every pixel with depth greater than *nearSharp* will have fuzziness of 0 (in focus), and every pixel in between *nearFuzzy* and *nearSharp* will have fuzziness linearly interpolated.

*nearSharp* must always be greater than *nearFuzzy*; otherwise you will get unexpected results.

When *nearSharp* is equal to *nearFuzzy*, the result will always be zero.

# edgePostExtractFarFuzziness

Calculates far fuzziness for depth of field

## Definition

```
#include <edgepost_spu.h>
void edgePostExtractFarFuzziness (
    void* output,
    const void* linearDepth,
    const void* nearFuzziness,
    uint32_t count,
    float farSharp,
    float farFuzzy,
    float maxFuzziness
)
```

## Calling Conditions

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

<i>output</i>	Pointer to output pixels
<i>linearDepth</i>	Pointer to input image with depth information
<i>nearFuzziness</i>	Pointer to secondary input image with near fuzziness information
<i>count</i>	Number of pixels
<i>farSharp</i>	Normalized depth value, every pixel with depth smaller will be in focus
<i>farFuzzy</i>	Normalized depth value, every pixel with depth greater will be out of focus
<i>maxfuzziness</i>	Maximum fuzziness value

## Return Values

None.

## Description

This function will generate a combined fuzziness buffer from a depth buffer, a near fuzziness buffer.

*linearDepth* is a pointer to a single-channel floating-point image with linear depth information.

*nearFuzziness* is a pointer to a single-channel floating-point image with near fuzziness as generated by [edgePostExtractNearFuzziness\(\)](#).

Every pixel with depth greater than *farFuzzy* will have fuzziness of 1 (out of focus), every pixel with depth less than *farSharp* will have fuzziness of 0 (in focus), and every pixel in between *farSharp* and *farFuzzy* will have fuzziness linearly interpolated.

*farSharp* must always be less than *farFuzzy*; otherwise you will get unexpected results. When *farSharp* is equal to *farFuzzy*, the result will be zero.

---

# edgePostMotionblur

---

Calculates motion blur on an input image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostMotionblur(
    void* output,
    const void* input,
    const void* motion,
    uint32_t stride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input color image
<i>motion</i>	Pointer to secondary input image with motion vectors
<i>stride</i>	Stride of input color image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

This function calculates motion blur on an input image. Motion vectors for each pixel are extracted from the motion buffer passed as a parameter.

The motion buffer must be an argb8 image where R is the X component of motion vector, and G is the Y component.

Input image must be converted into a four-channel floating-point buffer before being passed to this function. You can use `edgePostConvertArgb8ToFloats` to accomplish that.

The output image format will be argb8.

This function will calculate a motion blurred version of the input image. To do this it will sample 16 times along the motion vector for each pixel. The maximum sample distance is 16, so borders of 16x16 are needed for this function to work correctly.

The alpha of the output image will be `max( motionAmount, original.alpha )`. The maximum is taken so that you can feed the output of depth of field to the Motion Blur function and have a combined result that you can use to blend on top of the original image.

---

# edgePostMotionblur\_FX16

---

Calculates Motion Blur on an input image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostMotionblur_FX16(
    void* output,
    const void* input,
    const void* motion,
    uint32_t stride,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input color image
<i>motion</i>	Pointer to secondary input image with motion vectors
<i>stride</i>	Stride of input color image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

This function calculates Motion Blur on an input image. Motion vectors for each pixel are extracted from the motion buffer passed as a parameter.

The motion buffer must be an argb8 image where R is X component of motion vector, and G is Y component.

Input image must be converted into a four-channel floating-point buffer before being passed to this function. You can use `edgePostConvertArgb8ToFloats` to accomplish that.

The output image format will be fx16.

This function will calculate a motion blurred version of the input image. To do this it will sample 16 times along the motion vector for each pixel. The maximum sample distance is 16, so borders of 16x16 are needed for this function to work correctly.

The alpha of the output image will be `max( motionAmount, original.alpha )`. The maximum is taken so that you can feed the output of depth of field to the Motion Blur function and have a combined result that you can use to blend on top of the original image.



---

# edgePostMakeMaskFromFloats

---

Creates a mask image from a single-channel floating-point image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostMakeMaskFromFloats (
    void* output,
    const void* input,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input color image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

This function converts a single-channel floating-point image into an 8-bit per pixel image.

---

# edgePostApplyMaskFX16

---

Modulates an fx16 image by a mask image.

## Definition

---

```
#include <edgepost_spu.h>
void edgePostApplyMaskFX16 (
    void* output,
    const void* input,
    const void* mask,
    uint32_t count
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>output</i>	Pointer to output pixels
<i>input</i>	Pointer to input color image
<i>mask</i>	Pointer to the mask image
<i>count</i>	Number of pixels

## Return Values

---

None.

## Description

---

This function multiplies pixels of a fx16 image by the mask (an 8-bit image).

# PPU Functions

---

# edgePostInitializeWorkload

---

Prepares a workload for execution.

## Definition

---

```
#include <edgepost_ppu.h>
void edgePostInitializeWorkload(
    EdgePostWorkload* pWorkload,
    const EdgePostProcessStage* pStages,
    uint16_t stageCount
)
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>pWorkload</i>	Workload to be initialized
<i>pStages</i>	Pointer to the effect chain
<i>stageCount</i>	Size of the effect chain

## Return Values

---

None.

## Description

---

Call this function to prepare a workload for execution.

The workload will be initialized to execute the effect chain, which is passed as a parameter.

Always call this function before starting execution of a workload, for example every frame.

## Notes

---

After this function is called, the effect chain should be treated as read-only – at least until the effect chain has been fully executed, because it will be accessed by multiple SPUs.

---

# edgePostIsWorkloadFinished

---

Returns whether a workload is ended or not.

## Definition

---

```
#include <edgepost_ppu.h>
bool edgePostIsWorkloadFinished( EdgePostWorkload* pWorkload )
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>pWorkload</i>	Pointer to an <a href="#">EdgePostWorkload</a> to be tested
------------------	---

## Return Values

---

Returns `true` if workload is finished; otherwise returns `false`.

## Description

---

The function returns either `true` or `false` depending on whether the workload has fully executed or not. The parameter *pWorkload* must point to an initialized workload in order to get the proper result.

## See Also

---

[edgePostStallForWorkload](#)

---

# edgePostStallForWorkload

---

Stalls the PPU until a workload has finished executing.

## Definition

---

```
#include <edgepost_ppu.h>
void edgePostStallForWorkload( EdgePostWorkload* pWorkload )
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>pWorkload</i>	Pointer to an initialized workload
------------------	------------------------------------

## Return Values

---

None.

## Description

---

This function stalls the PPU until the workload passed as parameter has been fully executed.

## Notes

---

This function will keep calling [edgePostIsWorkloadFinished\(\)](#) and sleep for a small amount of time until [edgePostIsWorkloadFinished\(\)](#) returns true. So it is actually running a busy loop.

By design, Edge Post does not use any OS or SPURS primitive for synchronization. The intent is to keep the code as modular as possible. However, this will not stop users of the library from, for example, using an OS mutex to roll out their own synchronization mechanism.

## See Also

---

[edgePostIsWorkloadFinished](#)

---

# edgePostSelectTileSize

---

Utility function used to select suitable tile dimensions.

## Definition

---

```
#include <edgepost_ppu.h>
uint32_t edgePostSelectTileSize
(
    const uint32_t* pCandidates,
    uint32_t numCandidates,
    const EdgePostImage* pImages,
    uint32_t numImages,
    uint32_t heapSize
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>pCandidates</i>	An array of tile size candidates
<i>numCandidates</i>	Number of candidates
<i>pImages</i>	An array of images that need to be tiled
<i>numImages</i>	Number of images
<i>heapSize</i>	Amount of local store dedicated to tile data

## Return Values

---

Suitable candidate index or -1 if no candidate has been found and asserts are disabled.

## Description

---

This function can be used to select correct tile size for input/output image that will fit into local store. The input array of [EdgePostImage](#) is structured so that the first entry describes the output image size; each remaining entry is treated as an input image.

The last parameter tells the function how much space will be available to the runtime and must always be kept in sync with runtime code.

The candidate array contains potential tile sizes for the output image and must be of size *numCandidates* \* 2. The function will return the index of the selected candidate.

## Notes

---

If no suitable candidate can be found, the function will assert.

# edgePostSetupStage

Utility function to set up a stage of an effect chain.

## Definition

```
#include <edgepost_ppu.h>
void edgePostSetupStage (
    EdgePostProcessStage* pStage,
    const EdgePostImage* pImages,
    uint32_t numImages,
    const void* pEffectCode = 0,
    uint32_t effectCodeSize = 0,
    uint32_t effectCodeDmaSize = 0,
    const void* pStageParameter = 0,
    uint32_t stageParameterSize = 0,
    const uint32_t* pCandidates = 0,
    uint32_t numCandidates = 0,
    uint32_t totalHeapSize = EDGEPOST_DEFAULT_HEAP_SIZE
);
```

## Calling Conditions

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

<i>pStage</i>	Pointer to an <a href="#">EdgePostProcessStage</a> to be filled in
<i>pImages</i>	Pointer to an array of output/input images
<i>numImages</i>	Number of input/output images
<i>pEffectCode</i>	Pointer to some binary data in main memory
<i>effectCodeSize</i>	Amount of Local Store to be allocated for the effect code
<i>effectCodeDmaSize</i>	Size of the DMA for the effect code
<i>pStageParameter</i>	Pointer to the stage parameter area
<i>stageParameterSize</i>	Size in bytes of the stage parameter area
<i>pCandidates</i>	Pointer to candidate tile sizes for tile size selection
<i>numCandidates</i>	Number of candidates in the candidate array
<i>totalHeapSize</i>	Size of heap in Local Store that will be handed over to Edge Post

## Return Values

None.

## Description

This function fills relevant fields of an [EdgePostProcessStage](#) based on input parameters.

*pEffectCode* points to an optional area of SPU code to be DMAed just before the effect starts, *effectCodeSize* is the amount of memory to be reserved for the effect code, and *effectCodeDmaSize* is the size of the DMA to move the effect code into Local Store. For example *effectCodeDmaSize* could be the size for the code + data segment, whereas *effectCodeSize* could be code + data + bss.



If *pEffectCode* is NULL it is assumed that the user will provide effect code by their own means; in this case *effectCodeDmaSize* is ignored. If *effectCodeSize* is non-zero, an area of Local Store will be allocated anyway but no DMA is issued.

An optional pointer to a parameter area that will be DMAed to local store can be specified.

The difference between *pEffectCode* and the stage parameters is that *pEffectCode* will be cached in Local Store if possible whereas parameters are always read into Local Store via DMA.

If *pStageParameter* is NULL, no parameters are read and *stageParameterSize* is ignored.

This function will also try to find suitable tile sizes for the specified input/output image set by calling [edgePostSelectTileSize\(\)](#). If the pointer to *pCandidates* is NULL, this function will try to use a default set of candidates that works for 720p images (and down-sampled version of 720p). If you have different requirements you will have to specify your own set of candidates.

The default set of tile candidates is as follows:

```
static const uint32_t kDefaultTileCandidates[] =
{
    160, 90,
    80, 90,
    80, 60,
    80, 45,
    80, 30,
    80, 15,
    40, 22,
    20, 11,
};
```

The final parameter *totalHeapSize* holds the amount of space in local store that will be handed over to Edge Post and must always be kept in sync with the runtime value.

## Examples

---

```
EdgePostImage images[] =
{
    EdgePostImage( kEdgePostOutputTile, motionBlur, width, height, 4),
    EdgePostImage( kEdgePostInputTile, colors, width, height, 4,16,16, 4),
    EdgePostImage( kEdgePostInputTile, motionVectors, width, height, 4),
};
edgePostSetupStage( pStage, images, 3);
```

## Notes

---

Once this function has been called it is possible to modify remaining fields of [EdgePostProcessStage](#) if needed, such as RSX™ label address or user data values.

## See Also

---

[edgePostSelectTileSize](#)

# edgePostMlaaInitializeContext

Initializes an MLAA instance.

## Definition

```
#include <edgepost_mlaa_handler_ppu.h>
int edgePostMlaaInitializeContext
(
    EdgePostMlaaContext* context,
    uint32_t spus,
    CellSpurs* spurs,
    const uint8_t* priorities,
    uint32_t rsxLabel,
    void* memBlock,
    size_t memBlockSize
);
```

## Calling Conditions

No restrictions.

## Arguments

<i>context</i>	A pointer to a structure holding all context information for this instance
<i>spus</i>	Size of task set to be created
<i>spurs</i>	A SPURS instance
<i>priorities</i>	Pointer to an 8-element array containing the per-SPU priorities
<i>rsxLabel</i>	RSX™ label used to notify the GPU of competition
<i>memBlock</i>	Pointer to the working memory of this context
<i>memBlockSize</i>	Size of the working memory provided

## Return Values

The function returns CELL\_OK on success. (Other return codes come from the internally called functions.)

## Description

This function creates a valid [EdgePostMlaaContext](#) structure.

## Notes

If the function fails, the context must be passed to [edgePostMlaaDestroyContext\(\)](#) to prevent resource leaks.

The size of *memBlock* must be at least EDGE\_POST\_MLAA\_HANDLER\_SPU\_BUFFER\_SIZE and aligned according to EDGE\_POST\_MLAA\_HANDLER\_BUFFER\_ALIGN.

## See Also

[edgePostMlaaDestroyContext](#), [edgePostMlaaPrepareWithRelativeThreshold](#), [edgePostMlaaKickTasks](#), [edgePostMlaaWait](#)

---

# edgePostMlaaDestroyContext

---

Destroys an MLAA instance.

## Definition

---

```
#include <edgepost_mlaa_handler_ppu.h>
void edgePostMlaaDestroyContext
(
    EdgePostMlaaContext* context
);
```

## Calling Conditions

---

No restrictions.

## Arguments

---

<i>context</i>	A pointer to a structure holding all context information for this instance
----------------	--

## Return Values

---

The function returns `CELL_OK(0)` on success.

On failure, one of the following error codes is returned (these represent negative values and come exclusively from SPURS): `CELL_SPURS_TASK_ERROR_STAT` or `CELL_SPURS_TASK_ERROR_INVALID`.

## Description

---

Shuts down an MLAA instance and frees all memory associated with the context.

## Notes

---

This function will block until all tasks are shut down. Because the tasks must finish their current operation, it is recommended that you only call this function when no operation is ongoing. This will help prevent long stalls.

## See Also

---

[edgePostMlaaInitializeContext](#), [edgePostMlaaPrepareWithRelativeThreshold](#),  
[edgePostMlaaKickTasks](#), [edgePostMlaaWait](#)

# edgePostMlaaPrepareWithRelativeThreshold

Prepares an MLAA operation using relative thresholding as the edge detection method.

## Definition

```
#include <edgepost_mlaa_handler_ppu.h>
void edgePostMlaaPrepareWithRelativeThreshold
(
    EdgePostMlaaContext* context,
    const void* src,
    void* dst,
    uint32_t width,
    uint32_t height,
    uint32_t pitch,
    uint8_t base,
    uint8_t scale,
    uint32_t mode,
    uint32_t rsxLabelValue
);
```

## Calling Conditions

No restrictions.

## Arguments

<i>context</i>	A pointer to a structure holding all context information for this instance
<i>src</i>	Pointer to the source image buffer
<i>dst</i>	Pointer to the destination image buffer
<i>width</i>	Length of a scan-line in pixels
<i>height</i>	Number of scan-lines in source image
<i>pitch</i>	Pitch of source and destination image, in bytes
<i>base</i>	See Description
<i>scale</i>	See Description
<i>mode</i>	Operation mode
<i>rsxLabelValue</i>	Value to be written to the RSX™ label when the operation is completed

## Description

This function prepares the SPU tasks for an MLAA operation. The base and scale parameters correspond to the fields *parameter0* and *parameter1* of [EdgePostMlaaTaskParameters](#), respectively.

## Notes

This function writes the parameter structures of the SPURS tasks. To prevent race conditions, call [edgePostMlaaWait\(\)](#) before calling this function, to ensure that the parameter structures are no longer in use.

## See Also

[edgePostMlaaInitializeContext](#), [edgePostMlaaDestroyContext](#),  
[edgePostMlaaKickTasks](#), [edgePostMlaaWait](#), [EdgePostMlaaTaskParameters](#)

---

# edgePostMlaaKickTasks

---

Starts a prepared MLAA operation by activating the SPU Tasks.

## Definition

---

```
#include <edgepost_mlaa_handler_ppu.h>
void edgePostMlaaKickTasks
(
    EdgePostMlaaContext* context
);
```

## Calling Conditions

---

No restrictions.

## Arguments

---

<i>context</i>	A pointer to a structure holding all context information for this instance
----------------	--

## Description

---

This function starts the SPU's with the parameters set by the last call to [edgePostMlaaPrepareWithRelativeThreshold\(\)](#).

## See Also

---

[edgePostMlaaInitializeContext](#), [edgePostMlaaDestroyContext](#),  
[edgePostMlaaPrepareWithRelativeThreshold](#), [edgePostMlaaWait](#)

# edgePostMlaaWait

---

Waits for a running MLAA operation to finish.

## Definition

---

```
#include <edgepost_mlaa_handler_ppu.h>
void edgePostMlaaWait
(
    EdgePostMlaaContext* context
);
```

## Calling Conditions

---

No restrictions.

## Arguments

---

<i>context</i>	A pointer to a structure holding all context information for this instance
----------------	--

## Description

---

This function blocks until it is safe to overwrite the task parameters with [edgePostMlaaPrepareWithRelativeThreshold\(\)](#).

## See Also

---

[edgePostMlaaInitializeContext](#), [edgePostMlaaDestroyContext](#),  
[edgePostMlaaPrepareWithRelativeThreshold](#), [edgePostMlaaKickTasks](#)

# SPU Callback Functions

---

# EdgePostPollCallback

---

Callback function called to check if there is an SPU yield request.

## Definition

---

```
#include <edgepost_framework_spu.h>
unsigned int EdgePostPollCallback();
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

None.

## Return Values

---

Returns 0 when no SPU yield request is pending.

Returns != 0 when SPU yield request is pending.

## Description

---

Callback function called to check if there is an SPU yield request.

## See Also

---

[EdgePostSpuConfig](#)



---

# EdgePostStageEnterCallback

---

Callback function called when the SPU is beginning to process an effect stage.

## Definition

---

```
#include <edgepost_framework_spu.h>
EdgePostTileCallback EdgePostStageEnterCallback (
    const EdgePostProcessStage* stage,
    void* effectCode
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>stage</i>	Pointer to current stage descriptor
<i>effectCode</i>	Pointer to allocated area for the effect code

## Return Values

---

Function pointer to be used for processing tiles.

## Description

---

This function is called every time the SPU starts working on an effect stage. It must return the callback function that Edge Post will call for every Tile to be processed.

*effectCode* and *effectCodeSize* have a meaning only if *stage->tileJobSize* is non-zero. This area of local store will be primed with whatever is at *stage->tileJobEa* if it is not NULL.

## See Also

---

[EdgePostProcessStage](#)

---

# EdgePostStageExitCallback

---

Callback function called when the SPU ends processing on the current stage.

## Definition

---

```
#include <edgepost_framework_spu.h>
void EdgePostStageExitCallback(
    const EdgePostProcessStage* stage,
    void* effectCode
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>stage</i>	Pointer to current stage descriptor
<i>effectCode</i>	Pointer to allocated area for the effect code

## Return Values

---

None.

## Description

---

This function is called when the SPU ends processing on the current stage. Note that this function can be called if the current stage has completely ended but also if the user chose to yield the SPU to higher priority tasks.

## See Also

---

[EdgePostProcessStage](#)

---

# EdgePostTileCallback

---

Callback function called for each tile to be processed.

## Definition

---

```
#include <edgepost_framework_spu.h>
void EdgePostTileCallback(
    EdgePostTileInfo* tileInfo
);
```

## Calling Conditions

---

Can be called from an interrupt handler.

Can be called from a thread (does not depend on interrupt-disabled or -enabled state).

Multithread safe.

## Arguments

---

<i>tileInfo</i>	Pointer to a structure containing information on current tile
-----------------	---

## Return Values

---

None.

## Description

---

This function is called by Edge Post for each tile to be processed.

## See Also

---

[EdgePostTileInfo](#)

# Constants

---

# Return Codes

---

List of return codes returned by Edge Post.

## Definition

---

Macro	Value	Description
EDGEPOST_WORKLOAD_ENDED	0	Normal termination of a workload.
EDGEPOST_WORKLOAD_PREEMPTED	1	A task/job/workload of higher priority wants to take control.
EDGEPOST_WORKLOAD_IDLE	2	Another SPU is finishing the current stage of a workload.