

PlayStation®Edge Postライブラリ リファレンス

© 2010 Sony Computer Entertainment Inc.
All Rights Reserved.
SCE Confidential

目次

はじめに	5
このドキュメントについて	6
データ型	8
EdgePostSourceTile	9
EdgePostProcessStage	11
EdgePostWorkload	13
EdgePostTileInfo	14
EdgePostSpuConfig	15
EdgePostImage	17
EdgePostMlaaContext	18
EdgePostMlaaTaskParameters	19
EdgePostMlaaMemoryLayout	21
EdgePostInplaceTransposeMemoryLayout	22
EdgePostInplaceTransposePostOpFunction	23
SPU関数	24
edgePostSetSpuConfig	25
edgePostRunWorkload	26
edgePostMlaaPass	28
edgePostInplaceTransposePass	29
edgePostTransposeInPlace	30
SPU処理関数	32
edgePostDownsample8	33
edgePostDownsample16	34
edgePostDownsampleF	35
edgePostNearestDownsample	36
edgePostDownsampleFloatMin	37
edgePostDownsampleFloatMax	38
edgePostUpsample8	39
edgePostUpsample16	40
edgePostUpsampleF	41
edgePostGauss7x1_8	42
edgePostGauss7x1F	43
edgePostGauss1x7_8	44
edgePostGauss1x7F	45
edgePostBloomCapture8	46
edgePostBloomCaptureFX16	47
edgePostTonemapFX16	48
edgePostAvgLuminanceFX16	49
edgePostModulate8	50
edgePostModulateFX16	51
edgePostConstantModulate8	52
edgePostAddSat8	53
edgePostAddSatFX16	54

edgePostBlend8.....	55
edgePostBlendFX16.....	56
edgePostPremultiplyFX16.....	57
edgePostCombine.....	58
edgePostExtractDepth.....	59
edgePostFloatToGrayscale.....	60
edgePostArgb8ToFloats.....	61
edgePostFloatsToArgb8.....	62
edgePostFX16ToFloats.....	63
edgePostFX16ToArgb8.....	64
edgePostFP16LoToFloats.....	65
edgePostFP16HiToFloats.....	66
edgePostFP16ToFloats.....	67
edgePostFloatsToFP16.....	68
edgePostFX16ToFP16.....	69
edgePostLogLuvToFloats.....	70
edgePostLogLuvToFX16.....	71
edgePostLogLuvToArgb.....	72
edgePostFloatsToLogLuv.....	73
edgePostFX16ToLogLuv.....	74
edgePostFloatsToLuv.....	75
edgePostFX16ToLuv.....	76
edgePostLuvToFloats.....	77
edgePostLuvToFX16.....	78
edgePostLuvToArgb.....	79
edgePostFP16LuvToFloats.....	80
edgePostFP16LuvToFX16.....	81
edgePostFloatsToFP16Luv.....	82
edgePostFX16ToFP16Luv.....	83
edgePostDof.....	84
edgePostDof_FX16.....	85
edgePostInitializeDofInputBuffer.....	86
edgePostExtractNearFuzziness.....	87
edgePostExtractFarFuzziness.....	88
edgePostMotionblur.....	89
edgePostMotionblur_FX16.....	90
edgePostMakeMaskFromFloats.....	91
edgePostApplyMaskFX16.....	92
PPU関数.....	93
edgePostInitializeWorkload.....	94
edgePostIsWorkloadFinished.....	95
edgePostStallForWorkload.....	96
edgePostSelectTileSize.....	97
edgePostSetupStage.....	98
edgePostMlaaInitializeContext.....	100
edgePostMlaaDestroyContext.....	102
edgePostMlaaPrepareWithRelativeThreshold.....	103
edgePostMlaaKickTasks.....	105

edgePostMlaaWait	106
SPUコールバック関数	107
EdgePostPollCallback.....	108
EdgePostStageEnterCallback.....	109
EdgePostStageExitCallback	110
EdgePostTileCallback	111
定数	112
リターンコード	113

はじめに

このドキュメントについて

目的

このドキュメントは、Edge ライブラリの Edge Post コンポーネントの API リファレンスです。このコンポーネントは、画像の後処理を SPU 上で実行する場合に使用します。

対象読者と前提条件

このドキュメントは、PlayStation®3 用の高性能アプリケーションを書こうとしている PlayStation®3 デベロッパのため書かれたものです。デベロッパは、以下のような対象を熟知していることを前提にしています。

C と C++

PlayStation®3 のハードウェア

SCE の標準ライブラリ関数

関連ドキュメントやその他のリソース

Edge ライブラリ

このリファレンスと、以下のドキュメントを併用することにより、Edge ライブラリの使用法やリファレンスについての完全な情報を得ることができます。

- 「PlayStation® Edge ライブラリ 概要」
- 「PlayStation® Edge ジオメトリライブラリ クイックスタートガイド」
- 「PlayStation® Edge ジオメトリライブラリ リファレンス」
- 「PlayStation® Edge オフラインツール用ジオメトリライブラリ リファレンス」
- 「PlayStation® Edge アニメーションライブラリ リファレンス」
- 「PlayStation® Edge オフラインツール用アニメーションライブラリ リファレンス」
- 「PlayStation® Edge Zlib ライブラリ リファレンス」
- 「PlayStation® Edge LZMA ライブラリ リファレンス」
- 「PlayStation® Edge LZO ライブラリ リファレンス」
- 「PlayStation® Edge DXT ライブラリ リファレンス」

SPA および EdgePostFilterGen ツール

Edge パッケージに含まれる「SPU パイプライン化アセンブラ (SPA) ユーザガイド」は、アセンブリ最適化ツールの SPA について記述したものです。

EdgePostFilterGen については、同じく Edge パッケージに含まれる「EdgePostFilterGen ユーザガイド」を参照してください。これは、単純な画像カーネル処理向け（ガウス画像フィルタなど）の SPA に最適化されたループを生成するコマンドラインユーティリティです。それらのループは Edge Post で使用できます。その出力形式は、SPA ツールで読み込み可能な SPA ファイルです。

表記法

このドキュメントでは、以下のような印刷上の表記法を使います。

規則	意味
等幅フォント	プログラミングコードおよびリテラル（処理命令、レジスタ名、データ型、イベント、ファイル名など）を表します。また、関数、構造体、マクロなどの名前を表すこともあります。
等幅フォント+太字	構造体や関数の定義の中でのみ、構造体や関数の名前を示します。
等幅フォント+斜体	引数、パラメータ、変数を表します。
青色+下線のテキスト	ハイパーリンクを表します（青色で表示されるのは、カラープリンタもしくはオンラインの場合だけです）。

データ型

EdgePostSourceTile

入力画像のタイル設定を記述する構造体

定 義

```
#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)));
```

メ ン バ

<i>addressEa</i>	メインメモリ内での画像の開始位置の実効アドレス
<i>pitch</i>	画像のピッチ
<i>width</i>	画像内の 1 つのタイルの幅 (ピクセル)
<i>height</i>	画像内の 1 つのタイルの高さ (ピクセル)
<i>borderWidth</i>	タイルの左右の辺に必要なボーダーの幅
<i>borderHeight</i>	タイルの上下の辺に必要なボーダーの高さ
<i>bytesPerPixel</i>	ピクセル形式のサイズ (バイト)
<i>multiplier</i>	1 つのタイルのローカルストア占有率を計算する際に使用される乗数
<i>type</i>	タイルの種類 (入力、出力のいずれか)

解 説

これは、入力画像とそのタイル設定を記述するために使用される構造体です。

addressEa には、画像全体の 16 バイトアライン開始アドレスが格納されます。*pitch* は画像のピッチであり、通常は画像全体の幅に 1 ピクセル当たりのバイト数を掛けたものになります。

width と *height* には、この画像で選択されたタイルサイズが格納されます。これは画像全体の寸法ではありません。

borderWidth と *borderHeight* はタイルの周りのボーダーの必要サイズです。タイルの処理時にはこれらのボーダーがローカルストア内に存在している必要があります。タイルのボーダーは、あるピクセルの計算結果がその周囲のピクセルに依存する状況が発生するたびに必要となります。現在の中心からもっとも遠くのピクセルへのアクセスが可能になるだけの幅/高さを持つボーダーを指定すべきです。たとえば、水平 9x1 ガウスブラーを実行するには、少なくとも 4 ピクセル分のボーダーが必要になります。

borderWidth はピクセル数で指定されますが、その実際のサイズ ($\text{borderWidth} * \text{bytesPerPixel}$ バイト) は 16 バイトでアラインされている必要があります。*borderHeight* は (タイルの上下の) スキャンライン数で指定されます。

タイルの実際のピッチは、 $(\text{borderWidth} * 2 + \text{width}) * \text{bytesPerPixel}$ として計算できることに注意してください。この値は 16 バイトでアラインされており、かつ 16KB を決して超えないようにする必要があります。なぜなら、各タイルラインは単一の DMA 処理を使って転送されるからです。

bytesPerPixel には各ピクセルのサイズ（バイト）が格納されます（ARGB 画像の場合は通常 4 バイト）。

multiplier は、SPU ローカルストア内におけるタイルの実際の占有率を計算する際に使用される係数です。これを使えば、タイルの使用前に特定の変換をタイルに適用できるだけの余裕のある領域を割り当てることができます。

SPU ローカルストア内でのタイルの占有率は、次のように計算できます。

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

最後に、*type* はタイルが入力タイル、出力タイルのいずれとして処理されるかを指示します。

関 連 項 目

[EdgePostProcessStage](#)

EdgePostProcessStage

エフェクトチェーン内の1つのステージを記述する構造体

定義

```
#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)));
```

メンバ

<i>numTileX</i>	1行当たりのタイル数
<i>numtileY</i>	タイル行の数
<i>pad</i>	構造体のパディング（現在未使用）
<i>flags</i>	汎用フラグ
<i>rsxLabelAddress</i>	（オプション）RSX®ラベルのアドレス
<i>rsxLabelValue</i>	（オプション）RSX®ラベルの更新に使用される値
<i>effectCodeEa</i>	（オプション）エフェクトコードの実効アドレス
<i>effectCodeSize</i>	エフェクトコード用に確保すべきメモリの量
<i>effectCodeDmaSize</i>	エフェクトコードの DMA 転送のサイズ
<i>stageParametersEa</i>	ステージパラメータ領域の実効アドレス
<i>stageParametersSize</i>	ステージパラメータ領域のサイズ
<i>user0</i>	ユーザデータ
<i>user1</i>	ユーザデータ
<i>user2</i>	ユーザデータ
<i>user3</i>	ユーザデータ
<i>userDataI</i>	ユーザデータ
<i>userDataF</i>	ユーザデータ
<i>sources</i>	タイル記述子の配列

解 説

これは Edge Post の基礎となる構造体です。というのも、これには 1 つの処理ステージに関する情報がすべて格納されるからです。

記述された処理は、画像の各タイルに対して実行されます。*numTileX* と *numTileY* には、この特定の出力画像/入力画像セットに対するタイル数が格納されます。

flags には次のような、ステージの汎用情報が格納されます。

表 1 ステージのフラグ

EDGE_POST_BREAKPOINT	タイル関数を呼び出す前にブレークポイント命令を実行します。このフラグは、デバッグ目的で選択的に使用できます。
EDGE_POST_WRITE_RSX_LABEL	このフラグが設定されると、Edge Post は処理の完了時に、 <i>rsxLabelValue</i> に指定された値を <i>rsxLabelAddress</i> に指定されたアドレスに転送します。

rsxLabelAddress は RSX@ラベルへのポインタ（オプション）です。この特定の処理が終了し、すべての出力データのメモリへの転送が完了すると、*rsxLabelValue* に指定した値でこのラベルが更新されます。このフィールドを使えば、結果が取得可能になったことを RSX@に知らせることができます。*effectCodeEa* は SPU のローカルストアにアップロードされて画像のタイルごとに実行される SPU コードの実効アドレス（オプション）、*effectCodeSize* はローカルストア内のエフェクトコードのサイズ、*effectCodeDmaSize* はコードの転送に必要となる DMA のサイズです。

これらのフィールドはオプションです。コードをロードする必要がない場合やユーザ自身がコードのロードを行う場合には、このアドレスをゼロに設定します。しかしアドレスがゼロに設定されてもサイズがゼロでなければ、Edge Post はやはりメモリの割り当てを行います。ただし、DMA の処理は一切行いません（その処理はユーザが行うものと仮定される）。したがって、この機能がまったく必要ない場合には、サイズもゼロに設定してください。

各エフェクトには、そのコードとともに転送されるパラメータ領域を割り当てることができます。この領域へのポインタが *stageParametersEa* に格納され、そのサイズが *stageParametersSize* に格納されます。

sources は、ステージの画像記述子（最大 4 つ）を含む配列です。詳細については「[EdgePostSourceTile](#)」を参照してください。

EdgePostWorkload

現在実行中のエフェクトチェーンの格納に使用される構造体

定 義

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

解 説

この構造体は常に 128 バイトでアラインされ、そのサイズは 16 バイトになります。これは、SPU からアトミック操作を使ってアクセスされ、同じエフェクトチェーンの異なるタイルを処理している多数の SPU 間の同期を取るために使用されます。

EdgePostTileInfo

ユーザ提供タイルコールバックへのパラメータとして渡される構造体

定 義

```
#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;
};
```

メ ン バ

<i>stage</i>	現在の処理ステージ記述子へのポインタ
<i>tile_x</i>	現在のタイルの X 座標
<i>tile_y</i>	現在のタイルの Y 座標
<i>tiles</i>	タイルデータへのポインタの配列。
<i>parameters</i>	ステージ固有パラメータへのポインタ

解 説

この構造体は、ユーザ提供タイル関数へのパラメータとして渡されます。

EdgePostSpuConfig

SPU 上で実行される Edge Post タイル処理フレームワークの初期化に使用される構造体

定 義

```
#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)));
```

メ ン バ

<i>heapStart</i>	Edge Post に割り当てられたローカルストア領域の開始位置
<i>heapSize</i>	Edge Post に割り当てられたローカルストア領域のサイズ
<i>controlDmaTag</i>	汎用転送に使用される DMA タグインデックス
<i>inputDmaTag</i>	DMA get に使用される DMA タグインデックス
<i>outputDmaTag</i>	DMA put に使用される DMA タグインデックス
<i>pollCallback</i>	イールドが必要かどうかを確認するために使用されるコールバック
<i>stageStartCallback</i>	新しいエフェクトステージの開始時に呼び出されるコールバック
<i>stageEndCallback</i>	現在のエフェクトステージの終了時に呼び出されるコールバック

解 説

Edge Postが動作する各SPU上では、この構造体のすべてのフィールドを初期化して [edgePostSetSpuConfig](#) にパラメータとして渡さないと、他の処理を一切行うことができません。
heapStart と *heapSize* は、Edge Post が内部的に使用する領域（SPU ローカルストア内）の開始位置とサイズに初期化する必要があります。
controlDmaTag、*inputDmaTag*、および *outputDmaTag* は、Edge Post 自身の内部転送用としてユーザが Edge Post に渡したい DMA タグです。
pollCallback は型 [EdgePostPollCallback](#) の関数へのポインタです。この関数はタイルの処理中にさまざまなタイミングで呼び出され、現在のSPU上で実行すべきより優先順位の高いワークロードに対するポーリングをユーザが行えるようにします。
stageStartCallback は型 [EdgePostStageStartCallback](#) の関数へのポインタです。この関数は、エフェクトチェーン内の新しいステージが開始されようとするたびに呼び出されます。
stageEndCallback は型 [EdgePostStageEndCallback](#) の関数へのポインタです。この関数は、現在処理中のステージが終了するたびに呼び出されます。

備 考

`heapSize` がタイルを格納するのに十分なサイズであることを、常に確認するようにしてください。この値は、PPU 上でタイルサイズ計算時に使用される値と同期が取れている必要があります。

EdgePostImage

PPU ヘルパー関数によって使用される構造体。入力または出力画像のメモリレイアウトを記述する

定 義

```
#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;
};
```

メ ン バ

<i>m_dir</i>	画像が入力、出力のいずれであることを示します
<i>m_ea</i>	画像が格納されているバッファの実効アドレス
<i>m_pitch</i>	画像のピッチ
<i>m_width</i>	画像の幅（ピクセル）
<i>m_height</i>	画像の高さ（ピクセル）
<i>m_pixelSize</i>	1つのピクセルのサイズ（バイト）
<i>m_borders</i>	各タイルで必要になるボーダー（ピクセル）
<i>m_multiplier</i>	ローカルストア占有率の計算時に使用されるスペース乗数

解 説

このPPU構造体は、いくつかのPPU関数（正しいタイルサイズを選択を支援する関数や [EdgePostProcessStage](#) 構造体のセットアップを行う関数）の入力として使用されます。

EdgePostMlaaContext

MLAA（形態学的アンチエイリアス）システムの状態を格納するため、PPU ヘルパー関数によって使われる構造体

定 義

```
#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;
}
```

メ ン バ

<i>spuCount</i>	使われる論理 SPU の数。タスクの数に対応しています。
<i>spurs</i>	使われる SPURS インスタンスへのポインタ。
<i>taskSet</i>	この MLAA インスタンスで使われる SPURS タスクセットへのポインタ。
<i>taskIds</i>	MLAAタスクのSPURSタスク IDの配列へのポインタ。このフィールドが有効なのは、 edgePostMlaaInitializeContext () を呼び出した後だけであり、主に内部用です。
<i>taskAttributes</i>	<i>spuCount</i> 個のタスク属性の配列へのポインタ。
<i>taskArguments</i>	EdgePostMlaaTaskParameters () の配列へのポインタ。このフィールドが有効なのは、 edgePostMlaaInitializeContext () を呼び出した後だけであり、主に内部用です。
<i>taskParameters</i>	<i>spuCount</i> 個のタスクパラメータの配列へのポインタ。
<i>saveConfigs</i>	<i>spuCount</i> 個のタスク保存設定へのポインタ。
<i>barrier</i>	MLAA および転置パスの間で SPU を同期するために使われる、SPURS バリアへのポインタ。
<i>directionLock</i>	128 バイトにアラインメントされたメモリ位置へのポインタ。
<i>rsxLabel</i>	処理完了後に RSX®をリリースするために使われるラベル。
<i>rsxLabelAddress</i>	RSX®ラベルのアドレス。
<i>tasksReady</i>	この値は、処理完了時に、SPU によって設定されます。

解 説

このPPU構造体は、MLAAのPPU側で、インスタンス情報を保存するために使われます。この構造体の設定には、[edgePostMlaaInitializeContext \(\)](#) を使うことをお勧めします。

EdgePostMlaaTaskParameters

MLAA SPU タスクにパラメータを渡すことを容易にするために使われる構造体

定 義

```
#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];
};
```

メ ン バ

<i>rsxLabelValue</i>	処理完了時に RSX®ラベルに書き込まれる値。
<i>rsxLabelAddress</i>	RSX®ラベルの実効アドレス。書き込む必要がない場合には NULL。
<i>taskCounterAddress</i>	SPU が完了するたびにインクリメントされるカウンタの実効アドレス。NULL に設定してもかまいません。
<i>imageAddress</i>	ソース画像バッファの実効アドレス。16 バイトにアラインメントする必要があります。128 バイトにアラインメントすることをお勧めします。
<i>destAddress</i>	デスティネーション画像バッファの実効アドレス。16 バイトにアラインメントする必要があります。128 バイトにアラインメントすることをお勧めします。 <i>imageAddress</i> と同じでもかまいません。
<i>barrierAddress</i>	<i>spuCount</i> 個の SPU 用に設定された SPURS バリアの実効アドレス。
<i>directionLockAddress</i>	128 バイトにアラインメントされ、ゼロに初期化された、4 バイトの XDR メモリ位置の実効アドレス。
<i>imageWidth</i>	処理対象のスキャンラインのピクセルの数。
<i>imageHeight</i>	画像の高さ。パディングを含みません。
<i>imagePitch</i>	画像バッファの水平ピッチ（バイト単位）。
<i>mode</i>	MLAA モード。「解説」を参照してください。
<i>spuId</i>	タスクグループ中のこの SPU の数。
<i>spuCount</i>	タスクグループのサイズ。
<i>parameter0</i>	エッジ検出のベース値。「解説」を参照してください。
<i>parameter1</i>	エッジ検出のスケール。「解説」を参照してください。
<i>reserved</i>	内部用に予約されています。

解 説

各 MLAA タスクは、この構造体のコピーを受け取ります。値の設定には通常、[edgePostMlaaPrepareWithRelativeThreshold\(\)](#) が使われます。

タスクの完了通知の方法としては、RSX®ラベル、XDR カウンタの 2 種類がサポートされています。
`rsxLabelAddress` が 0 でない場合には、SPU は、全 SPU との同期後に、`rsxLabelValue` を書き込みます。同じように、`taskCounterAddress` が 0 でない場合には、SPU は処理を完了して同期が済んだ後で、値をインクリメントします。このような値は SPU ごとに設定され、その書き込みは SPU の処理完了後に行われるので、一般に、これらのアドレスのうちの 1~2 個を 1 つの SPU だけで設定することに注意してください。

`imageAddress` の指すバッファのサイズは、`imagePitch*imageHeight` であるとみなされます。また、`destAddress` の指すバッファの高さは、128 で割り切れる必要があります。つまり、720p のバッファには、1280*768 ピクセルの領域が必要です。入力と出力に同じバッファを使っても、パフォーマンスが低下することはありません。

`directionLockAddress` の指すバッファは、転置パス中に繰り返し使われるので、偽共有を避けるため、固有のキャッシュラインに配置することをお勧めします。

パラメータ `parameter0` はピクセルしきい値の下限を定義します。`parameter1` は 0.8 固定小数点でエンコードされたスケールです。このスケールは、各ピクセルの RGB チャンネル中の最高の値をもつチャンネルに乗算されます。この値の最大値、および `parameter1` の下限は、このピクセルで使われるしきい値を定義します。

エッジ検出時のエッジ判定基準は、2 ピクセル間の任意のカラーチャンネルの絶対差が、その 2 ピクセルの最小しきい値より大きいことです。

MLAA には複数の動作方法があり、`mode` パラメータを使って選択できます。フラグは自由に組み合わせることができますが、役に立つのは一部の組み合わせだけです。サポートされている全フラグの一覧は、以下の通りです。

EDGE_POST_MLAA_MODE_ENABLED	このフラグが設定されていない場合、タスクは <code>imageAddress</code> から <code>destAddress</code> への単純なコピーを実行します。
EDGE_POST_MLAA_MODE_SHOW_EDGES	水平エッジを赤色、垂直エッジを緑色にマークします。
EDGE_POST_MLAA_MODE_SHUTDOWN	タスクを終了します。このモードを設定すると場合、MLAA の処理は実行されず、モード以外の全パラメータが未定義になります。
EDGE_POST_MLAA_MODE_SINGLE_SPU_TRANSPOSE	転置操作を 2 つの SPU でなく 1 つの SPU で実行します。このフラグは、XDR ピーク圧を減らして、レイテンシを増やす代わりに、SPU 全体の使用量を最小化します。
EDGE_POST_MODE_TRANSPOSE_64	転置ブロックサイズを、128*128 から 64*64 に変更します。これにより、幅の制限も変更され、パディングされた高さは 128 の倍数ではなく 64 の倍数に変更されますが、その分パフォーマンスは悪化します。

備 考

絶対差のしきい値は、`parameter1` を 0 に設定して、必要なしきい値を `parameter0` に設定することにより設定できます。これにより、オリジナルで使われているしきい値関数が設定されます。

EdgePostMlaaMemoryLayout

SPU MLAA 関数用の作業メモリ

定 義

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

解 説

SPU 上で MLAA 関数を呼び出す際の作業メモリは、すべてこの構造体の形で渡されます。
MLAA 関数の呼び出しを複数回行う際に、この構造体のコンテンツを保持しておく必要はありません。
また、保持してもパフォーマンス上得することはありません。

備 考

アプリケーションで使っているのが提供された MLAA SPURS タスクである場合、SPU MLAA 関数を直接呼び出す必要はなく、したがって、この構造体を使う必要もありません。
構造体のサイズは、ローカルストア上のサイズとほとんど同じです。

EdgePostInplaceTransposeMemoryLayout

SPU のその場での擬似転置送関数用の作業メモリ

定 義

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

解 説

SPU 上でその場での転置関数を呼び出す際の作業メモリは、すべてこの構造体の形で渡されます。
その場での転置関数の呼び出しを複数回行う際に、この構造体のコンテンツを保持しておく必要はありません。また、保持してもパフォーマンス上得することはありません。

備 考

構造体のサイズは、ローカルストア上のサイズとほとんど同じです。

EdgePostInplaceTransposePostOpFunction

転置中に二次的な処理を行う際に使われる関数型

定 義

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

メ ン バ

<i>param</i>	関数に渡されるユーザ定義のパラメータ
<i>data</i>	転置後のデータバッファへのポインタ
<i>dataSize</i>	<i>data</i> の指すバッファのサイズ (バイト単位)

解 説

この型の関数は、[edgePostTransposeInPlace\(\)](#)に渡すことができます。

備 考

この型の関数の目的は、DMA リクエストの完了を待っている間に SPU がストールしないように、簡単な操作を実行することです。転置操作時に隠すことのできる処理の目安は、*data* 中の 1 クワッドワードあたり約 8 サイクルです。

関 連 項 目

[edgePostTransposeInPlace](#)

SPU関数

edgePostSetSpuConfig

SPU タイル処理フレームワークの設定と初期化を行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフではありません。

引 数

config タイル処理フレームワークの設定情報

返 り 値

なし。

例

```
// ヒープ領域を割り当てます
const int kHeapSize = 1024 * 200;
void* pHeap = memalign( 128, kHeapSize );
EDGE_ASSERT( pHeap && "no memory" );

// edgePost の設定情報をセットアップします
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;

// 設定情報を設定します
edgePostSetSpuConfig ( &config );
```

備 考

SPU 上で Edge Post ワークロードを開始する前にならずこの関数を呼び出します。

関 連 項 目

[EdgePostSpuConfig](#)

edgePostRunWorkload

Edge Post ワークロードの実行を開始するか、あるいはその実行に参加する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフではありません。

引 数

workloadEa 初期化済み [EdgePostWorkload](#) の実効アドレス

返 り 値

次のいずれかのコードを返します。

マクロ	値	解説
EDGEPOST_WORKLOAD_ENDED	0	現在のワークロードの実行が終了しました。
EDGEPOST_WORKLOAD_PREEMPTED	1	優先順位の高い別のタスク、ジョブ、またはワークロードを実行する必要があります。
EDGEPOST_WORKLOAD_IDLE	2	ワークロードは完了していませんが、別の SPU がエフェクトチェーン内の現在のステージを完了するまで、現在の SPU は待機する必要があります。

解 説

この関数は、ある Edge Post ワークロードの実行を開始します。そのワークロードが他の SPU によってすでに処理されている場合には、この関数を呼び出した SPU は、他の SPU と処理を共有しながらその実行を支援します。

この関数がリターンする理由としては、次の 3 つが考えられます。

- エフェクトチェーンが完了した。
- 優先順位の高い別のタスク/ジョブが SPU の制御の取得を要求している。
- 別の SPU が、現在のエフェクトステージの最後のタイルの処理を終えようとしている。現在のステージが完了するまで、他の SPU は次のステージに進むことができません。

この関数をタイトループ内で呼び出し、ループ終了条件 `EDGEPOST_WORKLOAD_ENDED` の成立をチェックしてもかまいません。

あるいは、SPU は、この関数から `EDGEPOST_WORKLOAD_IDLE` が返された場合に同じ関数を再度呼び出す前に何らかの無関係な処理を行ったり、優先順位の高い別のワークロードに SPU をイールドしたりすることも可能です。

例

```
while (1)
{
// メインの実行関数を呼び出します
    int ret = edgePostRunWorkload( workloadEa );

    if ( ret == EDGEPOST_WORKLOAD_PREEMPTED)
    {
// この SPU を他のプロセスと共有したい場合は、ここでイールドします
// たとえば、cellSpursYield() を呼び出します
    }

    if ( ret == EDGEPOST_WORKLOAD_ENDED)
    {
// ワークロードが完了しました。タスクを終了します
        break;
    }

// ワークロードは完了していませんが、この特定の SPU は現在アイドル状態になっています。
// 他の SPU が現在のステージの最後のタイルの処理を終えるまで、待機する必要があります
// あるからです。
}
```

edgePostMlaaPass

単一の水平・垂直 MLAA パスを実行する

定 義

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

呼 出 条 件

マルチスレッドセーフではありません。

引 数

<i>layout</i>	作業メモリへのポインタ
<i>parameter</i>	タスクパラメータへのポインタ
<i>pass</i>	水平パスの場合は 0、垂直パスの場合は 1
<i>first</i>	最初のパスでは 1、それ以外では 0
<i>baseDmaTag</i>	使用する最低 DMA タグ

返 り 値

なし

解 説

この関数は、*pass* パラメータに応じて、水平パスまたは垂直パスを実行します。垂直パスの場合、入力データが擬似転置形式である必要があります。これは、ソースバッファに対して [edgePostInplaceTransposePass \(\)](#) を実行すれば実現できます。

備 考

パフォーマンスを向上させるために、特に処理に使われる SPU が 3 個以上ある場合には、垂直パスを先に実行することをお勧めします。

この関数では、*baseDmaTag* から始まる 18 個の DMA タグを使います。このタグが [edgePostInplaceTransposePass \(\)](#) で使われるタグと重複しても問題ありません。

関 連 項 目

[edgePostInplaceTransposePass](#)

edgePostInplaceTransposePass

MLAA で使われるその場での疑似転置パスを実行する

定 義

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

呼 出 条 件

マルチスレッドセーフではありません。

引 数

<i>layout</i>	作業メモリへのポインタ
<i>parameter</i>	タスクパラメータへのポインタ
<i>first</i>	最初のパスでは 1、それ以外では 0
<i>baseDmaTag</i>	使用する最低 DMA タグ

返 り 値

なし

解 説

この関数は、*parameter* の *imageAddress* メンバの指すバッファから、*destAddress* メンバのよって指すバッファに、疑似転置を実行します。

備 考

MLAA 処理の最初のパスである場合には、以後のパスのために、*imageAddress* を *destAddress* に設定しておくことが重要です。

[edgePostTransposeInPlace\(\)](#) を直接呼び出す代わりにこの関数を使ってください。なぜならば、この関数は可能であれば 2 つ（理想的な数）の SPU 上で転置を実行するからです。この関数は、操作と無関係なタスクを含めて、あらゆるタスクから呼び出すことができます。

この関数は、*baseDmaTag* から始まる 3 個の DMA タグを使います。このタグが [edgePostMlaaPass\(\)](#) で使われるタグと重複しても問題ありません。

関 連 項 目

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

edgePostTransposeInPlace

32bpp 画像に対して 128x128 ピクセルの擬似転置を実行する

定 義

```
#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
)
```

呼 出 条 件

マルチスレッドセーフではありません。

引 数

<i>layout</i>	作業メモリへのポインタ
<i>sourceEa</i>	ソースバッファの実効アドレス
<i>destEa</i>	デスティネーションバッファの実効アドレス
<i>imagePitch</i>	バッファのピッチ。128 の倍数である必要がある
<i>imageHeight</i>	バッファの高さ。128 の倍数である必要がある
<i>spuId</i>	SPU の ID。0 または 1
<i>lockAddress</i>	メインメモリ上の 128 バイトにアラインメントされたバッファの実効アドレス。 詳しくは「備考」を参照
<i>postOpFunction</i>	転置処理の後で呼び出される関数へのポインタ、または NULL
<i>postOpParameter</i>	<i>postOpFunction</i> に渡されるユーザ定義のパラメータ、または NULL
<i>baseDmaTag</i>	使用する最低 DMA タグ

返 り 値

なし

解 説

この関数は、128x128 ピクセルブロックの中でピクセルを転置することにより、擬似逆画像を計算するために使われます。アルゴリズムは、128 ピクセルのセグメントを結合することにより、垂直ピクセル列を復元することができます。実際の転置は SPU 時間の約半分しか消費しないので、転置後のバッファに対して、DMA 転送を待っている間に操作を行うこともできます。

この関数は、*baseDmaTag* から始まる 3 個の DMA タグを使います。

備 考

処理は2つの部分に分かれており、順番に実行することも並列に実行することもできます。

720p バッファを2個のSPUで転置すると、1SPU当たり約400 μ s かかります。

`lockAddress` は、この関数の中で同期プリミティブとして使われます。このプリミティブはアトミック操作を使ってメモリを読み書きするので、キャッシュラインを他の用途に使わないことをお勧めします。また、`lockAddress` の指す値は、この関数の呼び出し時には0である必要があります。この関数はこの値を0のままにして終了します。

関 連 項 目

`edgePostTransposePass`

SPU処理関数

edgePostDownsample8

バイリニアダウンサンプリング

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>stride</i>	入力の 1 行のストライド（バイト）
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、画像の 2 行の 2x2 バイリニアダウンサンプリングを実行します。その結果、ダウンサンプリングされた 1 行が得られます。

input 画像と *output* 画像はどちらも、argb 32 ビットとしてフォーマットされます。

count は、ダウンサンプリング後の 1 行当たりのピクセル数を指定します。

1 行当たりの入力ピクセル数は、8 の倍数である必要があります。

edgePostDownsample16

バイリニアダウンサンプリング

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>stride</i>	入力の 1 行のストライド（バイト）
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、画像の 2 行の 2x2 バイリニアダウンサンプリングを実行します。その結果、ダウンサンプリングされた 1 行が得られます。

input 画像と *output* 画像はどちらも、argb 16 ビット/チャンネルとしてフォーマットされます（各チャンネルは符号なし整数）。

count は、ダウンサンプリング後の 1 行当たりのピクセル数を指定します。

1 行当たりの入力ピクセル数は、8 の倍数である必要があります。

edgePostDownsampleF

バイリニアダウンサンプリング

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>stride</i>	入力の 1 行のストライド（バイト）
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、画像の 2 行の 2x2 バイリニアダウンサンプリングを実行します。その結果、ダウンサンプリングされた 1 行が得られます。

input 画像と *output* 画像はどちらも、単一チャンネル浮動小数点画像としてフォーマットされます。

count は、ダウンサンプリング後の 1 行当たりのピクセル数を指定します。

1 行当たりの入力ピクセル数は、8 の倍数である必要があります。

edgePostNearestDownsample

ニアレストダウンサンプリング

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、画像の 2 行（ピクセルサイズは 4）に対して 2x2 バイリニアダウンサンプリングを実行します。
選択されるサンプルは常に、2x2 クワッドの左上になります。
1 行当たりの入力ピクセル数は、8 の倍数である必要があります。

edgePostDownsampleFloatMin

サンプルから最小値を選択することにより、浮動小数点画像のダウンサンプリングを行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>stride</i>	入力の 1 行のストライド（バイト）
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、1 チャンネル浮動小数点画像の 2 行に対して 2x2 ダウンサンプリングを実行し、同画像の 1 行を生成します。選択候補のサンプルは、4 つのサンプルのうちの最小値になります。
この関数を使えば、たとえば、デプス画像をダウンサンプリングし、カメラにもっとも近いサンプルが常に選択されるようにすることができます。

edgePostDownsampleFloatMax

サンプルから最大値を選択することにより、浮動小数点画像のダウンサンプリングを行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>stride</i>	入力の 1 行のストライド（バイト）
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、1 チャンネル浮動小数点画像の 2 行に対して 2x2 ダウンサンプリングを実行し、同画像の 1 行を生成します。選択候補のサンプルは、4 つのサンプルのうちの最大値になります。
この関数を使えば、たとえば、デプス画像をダウンサンプリングし、カメラからもっとも遠いサンプルが常に変更されるようにすることができます。

edgePostUpsample8

バイリニアアップサンプリング

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>istride</i>	入力の 1 行のストライド (バイト)
<i>ostride</i>	出力の 1 行のストライド (バイト)
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数はバイリニアアップサンプリングを実行します。
input は画像の 1 行であり、*output* は 2 行（幅は 2 倍）です。
この関数は呼び出されるたびに、前と後の入力行にアクセスする必要があります。したがって、入力タイルの各辺に沿ってかならず十分な幅のボーダーを確保するようにしてください。タイルの上下の辺では最低でも 1 行分のボーダーが、左右の辺では 1 ピクセル分のボーダーが、それぞれ必要となります。
count は 1 つの出力行のピクセル数であり、8 の倍数になっている必要があります。
この関数は 4 チャンネル画像（各チャンネルは 8 ビット）を処理します。

edgePostUpsample16

バイリニアアップサンプリング

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>istride</i>	入力の 1 行のストライド (バイト)
<i>ostride</i>	出力の 1 行のストライド (バイト)
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数はバイリニアアップサンプリングを実行します。
input は画像の 1 行であり、*output* は 2 行（幅は 2 倍）です。
この関数は呼び出されるたびに、前と後の入力行にアクセスする必要があります。したがって、入力タイルの各辺に沿ってかならず十分な幅のボーダーを確保するようにしてください。タイルの上下の辺では最低でも 1 行分のボーダーが、左右の辺では 1 ピクセル分のボーダーが、それぞれ必要となります。
count は 1 つの出力行のピクセル数であり、8 の倍数になっている必要があります。
この関数は 4 チャンネル画像（各チャンネルは 16 ビット）を処理します。

edgePostUpsampleF

バイリニアアップサンプリング

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>istride</i>	入力の 1 行のストライド (バイト)
<i>ostride</i>	出力の 1 行のストライド (バイト)
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数はバイリニアアップサンプリングを実行します。
input は画像の 1 行であり、*output* は 2 行（幅は 2 倍）です。
この関数は呼び出されるたびに、前と後の入力行にアクセスする必要があります。したがって、入力タイルの各辺に沿ってかならず十分な幅のボーダーを確保するようにしてください。タイルの上下の辺では最低でも 1 行分のボーダーが、左右の辺では 1 ピクセル分のボーダーが、それぞれ必要となります。
count は 1 つの出力行のピクセル数であり、8 の倍数になっている必要があります。
この関数は単一チャンネル画像（チャンネルの形式は浮動小数点）を処理します。

edgePostGauss7x1_8

水平 7 ピクセル幅ガウスブラー

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	処理するピクセルの数
<i>weights</i>	パックされたガウスブラーウェイト

返 り 値

なし。

解 説

この関数は、入力ピクセル行に対して水平 7 ピクセル幅ガウスブラーを実行します。
weights には、フィルタ処理で使用するガウスウェイトが格納されます。
この関数が動作するには、3 ピクセル以上のボーダーが左右に必要となります。
この関数は 4 チャンネル画像（各チャンネルは 8 ビット）を処理します。

```
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

水平 7 ピクセル幅ガウスブラー

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	処理するピクセルの数
<i>weights</i>	パックされたガウスブラーウェイト

返 り 値

なし。

解 説

この関数は、入力ピクセル行に対して水平 7 ピクセル幅ガウスブラーを実行します。
weights には、フィルタ処理で使用するガウスウェイトが格納されます。
この関数が動作するには、3 ピクセル以上のボーダーが左右に必要となります。
この関数は単一チャンネル画像（チャンネルの形式は浮動小数点）を処理します。

```
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

垂直 7 ピクセル幅ガウスブラー

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>stride</i>	1 つの入力行のサイズ（バイト）
<i>count</i>	処理するピクセルの数
<i>weights</i>	パックされたガウスブラーウェイト

返 り 値

なし。

解 説

この関数は、入力ピクセル行に対して垂直 7 ピクセル幅ガウスブラーを実行します。
weights には、フィルタ処理で使用するガウスウェイトが格納されます。
この関数が動作するには、3 行以上のボーダーが上下に必要となります。
この関数は 4 チャンネル画像（各チャンネルは 8 ビット）を処理します。

```
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

垂直 7 ピクセル幅ガウスブラー

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>stride</i>	1 つの入力行のサイズ（バイト）
<i>count</i>	処理するピクセルの数
<i>weights</i>	パックされたガウスブラーウェイト

返 り 値

なし。

解 説

この関数は、入力ピクセル行に対して垂直 7 ピクセル幅ガウスブラーを実行します。
weights には、フィルタ処理で使用するガウスウェイトが格納されます。
この関数が動作するには、3 行以上のボーダーが上下に必要となります。
この関数は単一チャンネル画像（チャンネルの形式は浮動小数点）を処理します。

```
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

ソース画像からブルームカラーをキャプチャする

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	処理するピクセルの数
<i>exposureLevel</i>	輝度乗数
<i>minLuminance</i>	輝度しきい値（この値より大きいピクセルはブルームに寄与する）
<i>luminanceRangeRcp</i>	$1 / (\text{maxLuminance} - \text{minLuminance})$

返 り 値

なし。

解 説

この関数は、元のピクセル行から 1 行分のブルームカラーを生成します。
入力と出力はどちらも argb8 画像です。
ブルーム値は次のように計算されます。

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

edgePostBloomCaptureFX16

ソース画像からブルームカラーをキャプチャする

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	処理するピクセルの数
<i>exposureLevel</i>	輝度乗数
<i>minLuminance</i>	輝度しきい値（この値より大きいピクセルはブルームに寄与する）
<i>luminanceRangeRcp</i>	$1 / (\text{maxLuminance} - \text{minLuminance})$

返 り 値

なし。

解 説

この関数は、元のピクセル行から 1 行分のブルームカラーを生成します。
入力と出力はどちらも fx16 画像です。
ブルーム値は次のように計算されます。

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

edgePostTonemapFX16

画像にトーンマッピングを適用する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	処理するピクセルの数
<i>avgLuminance</i>	入力バッファの平均輝度レベル
<i>middleGray</i>	ミドルグレー値
<i>whitePoint</i>	ホワイトポイント値

返 り 値

なし。

解 説

この関数は、fx16 入力バッファに対してトーンマッピングとガンマ補正を行います。argb8 バッファを書き出します。

edgePostAvgLuminanceFX16

入力画像の平均輝度を計算する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	処理するピクセルの数
<i>avgLuminancePtr</i>	計算された平均輝度へのポインタ
<i>maxLuminancePtr</i>	計算された最大輝度へのポインタ

返 り 値

なし。

解 説

この関数は、入力 fx16 画像の平均/最大輝度レベルを計算します。

edgePostModulate8

2つの入力画像の相互変調を行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	1つ目の画像へのポインタ
<i>shuffle0</i>	1つ目の画像からロードされたピクセルに使用するシャッフルマスク
<i>input1</i>	2つ目の画像へのポインタ
<i>shuffle1</i>	2つ目の画像からロードされたピクセルに使用するシャッフルマスク
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、2つの argb8 ピクセル行を次のように掛け合わせます。

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

edgePostModulateFX16

2つの入力画像の相互変調を行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	1つ目の画像へのポインタ
<i>shuffle0</i>	1つ目の画像からロードされたピクセルに使用するシャッフルマスク
<i>input1</i>	2つ目の画像へのポインタ
<i>shuffle1</i>	2つ目の画像からロードされたピクセルに使用するシャッフルマスク
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、2つの fx16 ピクセル行を次のように掛け合わせます。

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

edgePostConstantModulate8

1つの画像を定数の係数で変調する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	1つ目の画像へのポインタ
<i>shuffle0</i>	1つ目の画像からロードされたピクセルに使用するシャッフルマスク
<i>constant</i>	2つ目のオペランド
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

この関数は、1行の argb8 ピクセルと指定された定数値を次のように掛け合わせます。

$$\text{result} = \text{shuffle0}(\text{input0}) \times \text{constant}$$

edgePostAddSat8

2つの画像を互いに足し合う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	1つ目の画像へのポインタ
<i>input1</i>	2つ目の画像へのポインタ
<i>multiplier</i>	整数乗数
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

2つの argb8 ピクセル行を互いに足し合わせます。

$result = input0 + input1 \times multiplier$

edgePostAddSatFX16

2つの画像を互いに足し合う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	1つ目の画像へのポインタ
<i>input1</i>	2つ目の画像へのポインタ
<i>multiplier</i>	整数乗数
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

2つの fx16 ピクセル行を互いに足し合わせます。

$result = input0 + input1 \times multiplier$

edgePostBlend8

2つの画像をブレンドする

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	1つ目の画像へのポインタ
<i>input1</i>	2つ目の画像へのポインタ
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

2つの argb8 ピクセル行をブレンドします。

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

edgePostBlendFX16

2つの画像をブレンドする

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	1つ目の画像へのポインタ
<i>input1</i>	2つ目の画像へのポインタ
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

2つの fx16 ピクセル行をブレンドします。

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

edgePostPremultiplyFX16

アルファチャンネルによる事前乗算を行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	入力画像へのポインタ
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

入力画像にそのアルファチャンネルを事前に掛けます。

```
result = input0 x input0.alpha
```

edgePostCombine

異なる 2 つの画像に含まれるピクセルを、シャッフルマスクを使って合成する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input0</i>	1 つ目の画像へのポインタ
<i>input1</i>	2 つ目の画像へのポインタ
<i>shuffleMask</i>	合成に使用されるシャッフルマスク
<i>count</i>	処理するピクセルの数

返 り 値

なし。

解 説

2 つのピクセル行を合成します（一度に 16 バイトを処理）。

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

edgePostExtractDepth

RSX®のデプスバッファをリニアデプスに変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	RSX®形式の入力デプスバッファへのポインタ
<i>count</i>	ピクセル数
<i>near</i>	ニアプレーンの場所
<i>far</i>	ファープレーンの場所

返 り 値

なし。

解 説

この関数を使えば、RSX®形式（D24S8）のデプスバッファを単一チャンネル浮動小数点画像（値 0 がニアプレーンに、値 1 がファープレーンにそれぞれマップされたもの）に変換できます。

edgePostFloatToGrayscale

浮動小数点単一チャンネル画像をグレースケール argb8 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

この関数は、1 チャンネル浮動小数点画像をグレースケール argb8 画像に変換します。これは主にデバッグ目的で提供されています。変換はその場で行えるため、入力ポインタと出力ポインタが同じであってもかまいません。

edgePostArgb8ToFloats

argb8 から浮動小数点 4 チャンネル画像への変換を行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力画像へのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

argb8 形式の入力画像（1 クセル当たり 4 バイト）を 4 チャンネル浮動小数点画像（1 ピクセル当たり 16 バイト）に変換します。
この関数はデータをその場で変換できるため、input と output が同じメモリ領域を指していてもかまいません。出力画像にはかならず十分なサイズの領域を割り当ててください。
この関数と入力タイルの *multiplier* モディファイアを組み合わせ使用すれば、タイルの形式をその場で、SPU コードからアクセスしやすい形式（各ピクセルのサイズが 16 バイト）に変換できます。

edgePostFloatsToArgb8

浮動小数点 4 チャンネル画像から argb8 画像への変換を行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力画像へのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 チャンネル浮動小数点画像を argb8 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFX16ToFloats

固定小数点 5:11 から浮動小数点 4 チャンネル画像への変換を行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力画像へのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

固定小数点 5:11 の入力画像を 4 チャンネル浮動小数点画像（1 ピクセル当たり 16 バイト）に変換します。この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指している場合でもかまいません。出力画像にはかならず十分なサイズの領域を割り当てるようにしてください。

edgePostFX16ToArgb8

固定小数点 5:11 から argb8 画像への変換を行う

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力画像へのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

固定小数点 5:11 の入力画像を argb8 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFP16LoToFloats

各 32 ビットワードの下位 FP16 を単一チャンネル浮動小数点画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

入力ピクセルの形式は、サイズが 4 バイトのものであるべきです。データの上位 16 ビットは破棄されます。データの下位 16 ビットが FP16 から 32 ビット浮動小数点に変換されます。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFP16HiToFloats

各 32 ビットワードの上位 FP16 を単一チャンネル浮動小数点画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

入力ピクセルの形式は、サイズが 4 バイトのものであるべきです。データの下位 16 ビットは破棄されます。データの上位 16 ビットが FP16 から 32 ビット浮動小数点に変換されます。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFP16ToFloats

4 チャンネル FP16 画像を FP32 画像に変換します

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 チャンネル FP16 画像を FP32 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFloatsToFP16

4 チャンネル FP32 画像を FP16 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 チャンネル FP32 画像を FP16 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFX16ToFP16

4 チャンネル fx16 (5:11) 画像を FP16 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 チャンネル fx16 画像を FP16 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostLogLuvToFloats

4 バイト Log(L)uv 画像を 4 チャンネル浮動小数点画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 バイト Log(L)uv 画像を 4 チャンネル浮動小数点画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostLogLuvToFX16

4 バイト Log(L)uv 画像を 4 チャンネル固定小数点 5:11 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 バイト Log(L)uv 画像を 4 チャンネル固定小数点 5:11 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostLogLuvToArgb

4 バイト Log(L)uv 画像を argb8 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 バイト Log(L)uv 画像を argb8 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFloatsToLogLuv

4 チャンネル浮動小数点画像を Log(L)uv 画像に変換します

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 チャンネル浮動小数点画像を Log(L)uv 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFX16ToLogLuv

4 チャンネル固定小数点 5:11 画像を Log(L)uv 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 チャンネル固定小数点 5:11 を Log(L)uv 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFloatsToLuv

4 チャンネル浮動小数点画像を Luv 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 チャンネル浮動小数点画像を Luv 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFX16ToLuv

FX16 画像を Luv 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

FX16 画像を Luv 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostLuvToFloats

Luv 画像を 4 チャンネル浮動小数点画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

Luv 画像を 4 チャンネル浮動小数点画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostLuvToFX16

Luv 画像を FX16 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

Luv 画像を FX16 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostLuvToArgb

Luv 画像を argb8 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

Luv 画像を argb8 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFP16LuvToFloats

FP16Luv 画像を 4 チャンネル浮動小数点画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

FP16Luv 画像を 4 チャンネル浮動小数点画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFP16LuvToFX16

FP16Luv 画像を fx16 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

FP16Luv 画像を fx16 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFloatsToFP16Luv

4 チャンネル浮動小数点画像を FP16Luv 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

4 チャンネル浮動小数点画像を FP16Luv 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostFX16ToFP16Luv

fx16 画像を FP16Luv 画像に変換する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力ピクセルへのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

fx16 画像を FP16Luv 画像に変換します。
この関数はデータをその場で変換できるため、*input* と *output* が同じメモリ領域を指していてもかまいません。

edgePostDof

入力画像のフィールドの深度を計算する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力画像へのポインタ
<i>stride</i>	入力画像のストライド（ピクセル）
<i>count</i>	ピクセル数
<i>tapOffsetTable</i>	DOF フィルタのタップオフセットを含む 16 個の <code>vec_float4</code> から成るテーブル

返 り 値

なし。

解 説

この関数は、入力画像の被写界深度を計算します。
`input` は、ボケ値を表すアルファチャンネルを持つ 4 チャンネル浮動小数点画像である必要があります。`edgePostDofPremultiply` を使えば、画像を正しい形式に変換できます。
`tapOffsetTable` は、ブラーカーネルの適用時に取得されるサンプルの周囲分布を定義する 16 個のオフセットを含むテーブルへのポインタです。各エントリの最初の `float` が X オフセット、2 番目の `float` が Y オフセットであり、残りの値は使用されません。また、このテーブル内のオフセットによってブラーの最大半径も定義されますが、さらにそれから、入力タイルに必要なボーダーのサイズの上限も決まります。
デフォルトのオフセットテーブルは、次のグローバル変数によって提供されています。
`extern const vec_float4 g_defaultDofTaps[];`
このテーブルを使用する場合には、タイルのボーダーが左右 8 以上、上下 7 以上に設定されている必要があります。
この関数の出力は、1 つの `argb8` ピクセル行になります。アルファにはボケ値が設定されます。この値は、ブラーがかかっていない元の画像とブレンドする際の係数として使用できます。

edgePostDof_FX16

入力画像のフィールドの深度を計算する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力画像へのポインタ
<i>stride</i>	入力画像のストライド（ピクセル）
<i>count</i>	ピクセル数
<i>tapOffsetTable</i>	DOF フィルタのタップオフセットを含む 16 個の <code>vec_float4</code> から成るテーブル

返 り 値

なし。

解 説

この関数は、入力画像の被写界深度を計算します。
`input` は、ボケ値を表すアルファチャンネルを持つ 4 チャンネル浮動小数点画像である必要があります。`edgePostDofPremultiply` を使えば、画像を正しい形式に変換できます。
`tapOffsetTable` は、ブラーカーネルの適用時に取得されるサンプルの周囲分布を定義する 16 個のオフセットを含むテーブルへのポインタです。各エントリの最初の `float` が X オフセット、2 番目の `float` が Y オフセットであり、残りの値は使用されません。また、このテーブル内のオフセットによってブラーの最大半径も定義されますが、さらにそれから、入力タイルに必要なボーダーのサイズの上限も決まります。
デフォルトのオフセットテーブルは、次のグローバル変数によって提供されています。
`extern const vec_float4 g_defaultDofTaps[];`
このテーブルを使用する場合には、タイルのボーダーが左右 8 以上、上下 7 以上に設定されている必要があります。
この関数の出力は、1 つの `fx16` ピクセル行になります。アルファにはボケ値が設定されます。この値は、ブラーがかかっていない元の画像とブレンドする際の係数として使用できます。

edgePostInitializeDofInputBuffer

後続の [edgePostDof\(\)](#) 呼び出しで使用する入力画像を準備する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>colors</i>	入力カラー画像へのポインタ
<i>fuzziness</i>	入力ボケ値画像へのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

このヘルパー関数は、argb8 カラー画像と「ボケ値」画像を入力として受け取ってそれらを合成し、[edgePostDof](#) への入力として適した単一の 4 チャンネル浮動小数点画像を生成します。
入力ボケ値画像は、[edgePostExtractFarFuzziness\(\)](#) を使って生成されたものでなければなりません。
この関数の動作は次のようになります。カラー画像が浮動小数点に変換され、その結果のアルファに、入力ボケ値を使って値が設定されます。

edgePostExtractNearFuzziness

被写界深度の近傍ボケ値を計算する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	デプス情報を含む入力画像へのポインタ
<i>count</i>	ピクセル数
<i>nearFuzzy</i>	正規化されたデプス値。デプスがこれより小さいピクセルはすべて、ピントが合わない状態になります
<i>nearSharp</i>	正規化されたデプス値。デプスがこれより大きいピクセルはすべて、ピントが合った状態になります
<i>maxfuzziness</i>	最大ボケ値

返 り 値

なし。

解 説

この関数は、デプスバッファから近傍ボケ値バッファを生成します。
入力画像と出力画像はどちらも 1 チャンネルの浮動小数点画像です。入力にはリニア化されたデプス値が含まれている必要があります。
デプスが *nearFuzzy* より小さいピクセルのボケ値は 1 (ピントが合っていない)、デプスが *nearSharp* より大きいピクセルのボケ値は 0 (ピントが合っている) となり、*nearFuzzy* と *nearSharp* の間にあるピクセルのボケ値は線形補間されます。
nearSharp は常に *nearFuzzy* よりも大きくなければなりません。そうでない場合は予想外の結果が得られます。*nearSharp* と *nearFuzzy* が等しい場合、結果は常にゼロになります。

edgePostExtractFarFuzziness

被写界深度の遠方ボケ値を計算する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>linearDepth</i>	デプス情報を含む入力画像へのポインタ
<i>nearFuzziness</i>	近傍ボケ値情報を含む 2 つ目の入力画像へのポインタ
<i>count</i>	ピクセル数
<i>farSharp</i>	正規化されたデプス値。デプスがこれより小さいピクセルはすべて、ピントが合った状態になります
<i>farFuzzy</i>	正規化されたデプス値。デプスがこれより大きいピクセルはすべて、ピントが合わない状態になります
<i>maxfuzziness</i>	最大ボケ値

返 り 値

なし。

解 説

この関数は、デプスバッファと近傍ボケ値バッファから合成ボケ値バッファを生成します。
linearDepth は、リニアデプス情報を含む単一チャンネル浮動小数点画像へのポインタです。
nearFuzziness は、近傍ボケ値を含む単一チャンネル浮動小数点画像 ([edgePostExtractNearFuzziness\(\)](#) で生成されたもの) へのポインタです。
デプスが *farFuzzy* より大きいピクセルのボケ値は 1（ピントが合っていない）、デプスが *farSharp* より小さいピクセルのボケ値は 0（ピントが合っている）となり、*farSharp* と *farFuzzy* の間にあるピクセルのボケ値は線形補間されます。
farSharp は常に *farFuzzy* より小さくなくてはなりません。そうでない場合は予想外の結果が得られます。*farSharp* と *farFuzzy* が等しい場合、結果は常にゼロになります。

edgePostMotionblur

入力画像のモーションブラーを計算する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力カラー画像へのポインタ
<i>motion</i>	モーションベクトルを含む 2 つ目の入力画像へのポインタ
<i>stride</i>	入力カラー画像のストライド
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

この関数は、入力画像のモーションブラーを計算します。各ピクセルのモーションベクトルは、パラメータとして渡されたモーションバッファから取得されます。

モーションバッファは argb8 画像で、その R がモーションベクトルの X 成分、G が Y 成分になっている必要があります。

入力画像は、この関数に渡す前に 4 チャンネル浮動小数点バッファに変換しておく必要があります。

edgePostConvertArgb8ToFloats を使えばそれを実現できます。

出力画像の形式は argb8 です。

この関数は、入力画像のモーションブラー版を計算します。これを行うために、ピクセルごとにモーションベクトルに沿って 16 回のサンプリングが行われます。サンプルの最大距離は 16 になります。したがって、この関数が正しく動作するには 16x16 のボーダーが必要となります。

出力画像のアルファは、 $\max(\text{motionAmount}, \text{original.alpha})$ になります。ここで最大値が取られているのは、被写界深度の出力をこのモーションブラー関数に入力し、その合成結果を元の画像の上にブレンドできるようにするためです。

edgePostMotionblur_FX16

入力画像のモーションブラーを計算する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力カラー画像へのポインタ
<i>motion</i>	モーションベクトルを含む 2 つ目の入力画像へのポインタ
<i>stride</i>	入力カラー画像のストライド
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

この関数は、入力画像のモーションブラーを計算します。各ピクセルのモーションベクトルは、パラメータとして渡されたモーションバッファから取得されます。

モーションバッファは argb8 画像で、その R がモーションベクトルの X 成分、G が Y 成分になっている必要があります。

入力画像は、この関数に渡す前に 4 チャンネル浮動小数点バッファに変換しておく必要があります。

edgePostConvertArgb8ToFloats を使えばそれを実現できます。

出力画像の形式は fx16 です。

この関数は、入力画像のモーションブラー版を計算します。これを行うために、ピクセルごとにモーションベクトルに沿って 16 回のサンプリングが行われます。サンプルの最大距離は 16 になります。したがって、この関数が正しく動作するには 16x16 のボーダーが必要となります。

出力画像のアルファは、`max(motionAmount, original.alpha)` になります。ここで最大値が取られているのは、被写界深度の出力をこのモーションブラー関数に入力し、その合成結果を元の画像の上にブレンドできるようにするためです。

edgePostMakeMaskFromFloats

単一チャンネル浮動小数点画像からマスク画像を作成する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力カラー画像へのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

この関数は、単一チャンネル浮動小数点画像を 8 ビット/ピクセルの画像に変換します。

edgePostApplyMaskFX16

fx16 画像をマスク画像で変調する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>output</i>	出力ピクセルへのポインタ
<i>input</i>	入力カラー画像へのポインタ
<i>mask</i>	マスク画像へのポインタ
<i>count</i>	ピクセル数

返 り 値

なし。

解 説

この関数は、fx16 画像のピクセルにマスク（8 ビット画像）を掛け合わせます。

PPU関数

edgePostInitializeWorkload

ワークロードの実行準備を整え

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>pWorkload</i>	初期化するワークロード
<i>pStages</i>	エフェクトチェーンへのポインタ
<i>stageCount</i>	エフェクトチェーンのサイズ

返 り 値

なし。

解 説

ワークロードの実行準備を整えるには、この関数を呼び出します。
ワークロードは、パラメータの1つとして渡されるエフェクトチェーンを実行するように初期化されます。
ワークロードの実行を開始する前にはかならずこの関数を呼び出します（フレームごとに呼び出すなど）。

備 考

この関数の呼び出しが完了したら、少なくともエフェクトチェーンの実行が完全に終了するまでは、エフェクトチェーンは読み取り専用として扱われるべきです。なぜなら、これには複数のSPUがアクセスするからです。

edgePostIsWorkloadFinished

ワークロードが終了したかどうかを返する

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

pWorkload テスト対象の [EdgePostWorkload](#) へのポインタ

返 り 値

ワークロードが終了している場合は true を返し、そうでない場合は false を返します。

解 説

この関数は、ワークロードの実行が完全に終了したかどうかに応じて true、false のいずれかを返します。適切な結果を得るには、パラメータ *pWorkload* が初期化済みのワークロードを指している必要があります。

関 連 項 目

[edgePostStallForWorkload](#)

edgePostStallForWorkload

ワークロードの実行が完了するまで PPU をストールする

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>pWorkload</i>	初期化済みワークロードへのポインタ
------------------	-------------------

返 り 値

なし。

解 説

この関数は、パラメータとして渡されたワークロードの実行が完全に終了するまで、PPU をストールします。

備 考

この関数は、[edgePostIsWorkloadFinished\(\)](#) から true が返されるまで、[edgePostIsWorkloadFinished\(\)](#) の呼び出しと短時間のスリープを繰り返します。したがって、これは実際にはビジーループを実行していることになります。

EdgePost は、OS や SPURS の同期用プリミティブを一切使用しない設計になっています。その意図は、コードのモジュール性を可能な限り維持することです。ただし、ライブラリのユーザが、たとえば OS のミューテックスを使って独自の同期メカニズムを実行したりするのが禁止されているわけではありません。

関 連 項 目

[edgePostIsWorkloadFinished](#)

edgePostSelectTileSize

適切なタイル寸法の選択に使用されるユーティリティ関数

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>pCandidates</i>	タイルサイズ候補の配列
<i>numCandidates</i>	候補の数
<i>pImages</i>	タイルに分割する必要がある画像の配列
<i>numImages</i>	画像の数
<i>heapSize</i>	タイルデータ専用のローカルストアの量

返 り 値

適した候補のインデックス。候補が見つからず、かつアサートが無効になっている場合は-1。

解 説

この関数を使えば、入力/出力画像のタイルサイズとして、ローカルストア内に収まる正しいサイズを選択することができます。

[EdgePostImage](#)の入力配列は、最初のエントリが出力画像のサイズを記述し、残りの各エントリが入力画像として処理される、といった構成になっています。

最後のパラメータはランタイムで利用可能な領域のサイズを関数に伝えますが、その値は常に、ランタイムコードと同期が取れている必要があります。

候補配列には出力画像のタイルサイズ候補が含まれますが、そのサイズは *numCandidates* * 2 でなければなりません。この関数は、選択された候補のインデックスを返します。

備 考

適する候補が見つからない場合、この関数はアサートを行います。

edgePostSetupStage

エフェクトチェーンの1つのステージをセットアップするためのユーティリティ関数

定 義

```
#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
);
```

呼 出 条 件

割り込みハンドラから呼び出し可能です。

スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。

マルチスレッドセーフです。

引 数

<i>pStage</i>	設定する EdgePostProcessStage へのポインタ
<i>pImages</i>	出力/入力画像の配列へのポインタ
<i>numImages</i>	入力/出力画像の数
<i>pEffectCode</i>	メインメモリ内の何らかのバイナリデータへのポインタ
<i>effectCodeSize</i>	エフェクトコード用として割り当てるローカルストアの量
<i>effectCodeDmaSize</i>	エフェクトコードの DMA のサイズ
<i>pStageParameter</i>	ステージパラメータ領域へのポインタ
<i>stageParameterSize</i>	ステージパラメータ領域のサイズ（バイト）
<i>pCandidates</i>	タイルサイズ選択に使用されるタイルサイズ候補へのポインタ
<i>numCandidates</i>	候補配列内の候補数
<i>totalHeapSize</i>	Edge Post に渡されるヒープ（ローカルストア内）のサイズ

返 り 値

なし。

解 説

この関数は、入力パラメータに基づいて [EdgePostProcessStage](#) の関連フィールドを設定します。

pEffectCode はエフェクトの開始直前に DMA 転送すべき SPU コードを含むオプション領域へのポインタ、*effectCodeSize* はエフェクトコード用に確保すべきメモリの量、*effectCodeDmaSize* はエフェクトコードをローカルストアに移動する際の DMA のサイズです。たとえば、*effectCodeDmaSize*

はコード+データセグメントのサイズになり、*effectCodeSize* はコード+データ+bss のサイズになるなどです。

pEffectCode が NULL の場合、エフェクトコードはユーザが独自の方法で提供するものとみなされます。この場合、*effectCodeDmaSize* は無視されます。*effectCodeSize* がゼロ以外であれば、ローカルストアの領域がいずれにしても割り当てられますが、DMA は発行されません。

ローカルストアに DMA 転送されるパラメータ領域へのポインタ（オプション）を指定することもできます。

pEffectCode とステージパラメータとの違いは、*pEffectCode* は可能であればローカルストア内にキャッシュされるのに対し、パラメータは常に DMA 経由でローカルストアに読み込まれる、という点にあります。

pStageParameter が NULL の場合、パラメータの読み込みは行われず、*stageParameterSize* は無視されます。

さらにこの関数は、指定された一連の入力/出力画像に適したタイルサイズの決定を試みるために、[edgePostSelectTileSize\(\)](#) の呼び出しも行います。*pCandidates* へのポインタが NULL の場合、この関数は 720p 画像（および 720p 画像のダウンサンプリング版）で正常に機能するデフォルト候補セットの使用を試みます。要件が異なる場合には、独自の候補セットを指定する必要があります。

デフォルトのタイル候補セットは次のとおりです。

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

最後のパラメータ *totalHeapSize* には Edge Post に渡す領域（ローカルストア内）のサイズを格納しますが、この値は常に、ランタイムの値と同期が取れている必要があります。

例

```
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);
```

備考

この関数の呼び出しが完了すると、[EdgePostProcessStage](#) の残りのフィールド（RSX®ラベルのアドレスやユーザデータの値など）を必要に応じて変更することが可能になります。

関連項目

[edgePostSelectTileSize](#)

edgePostMlaaInitializeContext

MLAA インスタンスを初期化する

定 義

```
#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
);
```

呼 出 条 件

制限はありません。

引 数

<i>context</i>	このインスタンスのコンテキスト情報のすべてを保持する構造体へのポインタ
<i>spus</i>	作成するタスクセットのサイズ
<i>spurs</i>	SPURS インスタンス
<i>priorities</i>	各 SPU の優先順位を含む 8 要素配列へのポインタ
<i>rsxLabel</i>	GPU に競合を通知するために使われる RSX®ラベル
<i>memBlock</i>	このコンテキストの作業メモリへのポインタ
<i>memBlockSize</i>	提供された作業メモリのサイズ

返 り 値

この関数は成功した場合には CELL_OK を返します（それ以外のリターンコードは、内部的に呼び出された関数のリターンコードです）。

解 説

この関数は、有効な [EdgePostMlaaContext](#) 構造体を作成します。

備 考

関数が失敗した場合には、リソースのリークを防ぐため、[edgePostMlaaDestroyContext\(\)](#) にコンテキストを渡す必要があります。

memBlock のサイズは、最低でも EDGE_POST_MLAA_HANDLER_SPU_BUFFER_SIZE 以上であり、EDGE_POST_MLAA_HANDLER_BUFFER_ALIGN にアラインメントされている必要があります。

関 連 項 目

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

edgePostMlaaDestroyContext

MLAA インスタンスを破棄する

定 義

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

呼 出 条 件

制限はありません。

引 数

<i>context</i>	このインスタンスのコンテキスト情報のすべてを保持する構造体へのポインタ
----------------	-------------------------------------

返 り 値

この関数は、成功した場合には CELL_OK (0) を返します。
失敗した場合には、CELL_SPURS_TASK_ERROR_STAT、または、CELL_SPURS_TASK_ERROR_INVALID のエラーコードのどちらかが返されます（SPURS から返される負の値）。

解 説

MLAA インスタンスを終了して、コンテキストに関連するメモリのすべてを解放します。

備 考

この関数は、全タスクが終了するまでブロックします。タスクは現在の操作を完了する必要があるので、この関数を呼び出すのは操作が行われていないときだけにすることをお勧めします。これにより、長時間のストールが防止されます。

関 連 項 目

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

edgePostMlaaPrepareWithRelativeThreshold

相対的な閾値化をエッジ検出法として利用した MLAA 操作を作成する

定 義

```
#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
);
```

呼 出 条 件

制限はありません。

引 数

<i>context</i>	このインスタンスのコンテキスト情報のすべてを保持する構造体へのポインタ
<i>src</i>	ソース画像バッファへのポインタ
<i>dst</i>	デスティネーション画像バッファへのポインタ
<i>width</i>	スキャンラインの長さ（ピクセル単位）
<i>height</i>	ソース画像中のスキャンラインの数
<i>pitch</i>	ソースおよびデスティネーション画像のピッチ（バイト単位）
<i>base</i>	「解説」を参照
<i>scale</i>	「解説」を参照
<i>mode</i>	操作モード
<i>rsxLabelValue</i>	操作完了時に RSX®ラベルに書き込まれる値

解 説

この関数は、MLAA操作のSPUタスクを作成します。baseおよびscaleパラメータは、それぞれ [EdgePostMlaaTaskParameters](#) の *parameter0* および *parameter1* フィールドに対応しています。

備 考

この関数は、SPURSタスクのパラメータ構造体の書き込みを行います。競合状態を防ぐため、この関数を呼ぶ前に [edgePostMlaaWait\(\)](#) を呼び出して、パラメータ構造体が使われていないことを確認するようにしてください。

関 連 項 目

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

edgePostMlaaKickTasks

SPU タスクをアクティブ化して、作成済みの MLAA 操作を開始する

定 義

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

呼 出 条 件

制限はありません。

引 数

<i>context</i>	このインスタンスのコンテキスト情報のすべてを保持する構造体へのポインタ
----------------	-------------------------------------

解 説

この関数は、最後に呼び出された [edgePostMlaaPrepareWithRelativeThreshold\(\)](#) によって設定されたパラメータを使ってSPUを起動します。

関 連 項 目

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

edgePostMlaaWait

実行中の MLAA 操作が完了するのを待つ

定 義

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

呼 出 条 件

制限はありません。

引 数

<i>context</i>	このインスタンスのコンテキスト情報のすべてを保持する構造体へのポインタ
----------------	-------------------------------------

解 説

この関数は、[edgePostMlaaPrepareWithRelativeThreshold\(\)](#)を使ってタスクパラメータを上書きしても問題がないようになるまでブロックします。

関 連 項 目

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

SPUコールバック関数

EdgePostPollCallback

SPU イールド要求の存在有無を確認するために呼び出されるコールバック関数

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

なし。

返 り 値

保留中の SPU イールド要求が存在しない場合は 0 を返します。
保留中の SPU イールド要求が存在する場合は 0 以外を返します。

解 説

SPU イールド要求の存在有無を確認するために呼び出されるコールバック関数。

関 連 項 目

[EdgePostSpuConfig](#)

EdgePostStageEnterCallback

あるエフェクトステージの処理を SPU が開始した際に呼び出されるコールバック関数

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>stage</i>	現在のステージ記述子へのポインタ
<i>effectCode</i>	エフェクトコードに割り当てられた領域へのポインタ

返 り 値

タイルの処理に使用される関数のポインタ。

解 説

この関数は、あるエフェクトステージの処理を SPU が開始するたびに呼び出されます。これは、処理対象のすべてのタイルに対して Edge Post が呼び出すコールバック関数を返す必要があります。
effectCode と *effectCodeSize* が意味を持つのは、*stage->tileJobSize* がゼロでない場合だけです。*stage->tileJobEa* NULL でない場合、そこに存在するデータがローカルストアのこの領域に格納されます。

関 連 項 目

[EdgePostProcessStage](#)

EdgePostStageExitCallback

現在のステージの処理を SPU が終了した際に呼び出されるコールバック関数

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>stage</i>	現在のステージ記述子へのポインタ
<i>effectCode</i>	エフェクトコードに割り当てられた領域へのポインタ

返 り 値

なし。

解 説

この関数は、現在のステージの処理を SPU が終了した際に呼び出されます。この関数の呼び出しは、現在のステージが完了した場合だけでなく、優先順位のより高いタスクへの SPU のイールドが選択された場合にも行われる点に注意してください。

関 連 項 目

[EdgePostProcessStage](#)

EdgePostTileCallback

タイルごとに呼び出されるコールバック関数

定 義

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

呼 出 条 件

割り込みハンドラから呼び出し可能です。
スレッドから呼び出し可能です（割り込みの有効/無効状態には依存しない）。
マルチスレッドセーフです。

引 数

<i>tileInfo</i>	現在のタイルに関する情報を含む構造体へのポインタ
-----------------	--------------------------

返 り 値

なし。

解 説

この関数は、Edge Post によって処理対象のタイルごとに呼び出されます。

関 連 項 目

[EdgePostTileInfo](#)

定数

リターンコード

Edge Post から返されるリターンコードの一覧

定 義

マクロ	値	解説
EDGEPOST_WORKLOAD_ENDED	0	ワークロードの正常終了。
EDGEPOST_WORKLOAD_PREEMPTED	1	優先順位の高いタスク/ジョブ/ワークロードが制御の取得を要求しています。
EDGEPOST_WORKLOAD_IDLE	2	別のSPUがワークロードの現在のステージを終了中です。