Freescale Semiconductor
Application Note

Document Number: AN3806
Rev. 0, 09/2009

MPC5500 Digital Image Processing Library 1

by: Abrahan Tezmol
Applications Engineer, Guadalajara,Mexico
Juan Ramos
Guadalajara,Mexico

1 Introduction

The MPC5500 Digital Image Processing Library 1 contains optimized functions for the MPC5500 family of processors with a signal processing engine (SPE APU) for commonly used digital image processing operations.

Contents

freescale™ semiconductor

2 Library Functions

The library contains the following functions:

spatial_filtering_int_2x2	Filtering of an 8-bit unsigned integer image based on spatial correlation with floating-point masks of size 2x2.
spatial_filtering_int_3x3	Filtering of an 8-bit unsigned integer image based on spatial correlation with floating-point masks of size 3x3.
image_binarize	Binary representation of an 8-bit unsigned integer image using a thresholding approach.
image_subtract	Subtraction of two 8-bit unsigned integer images.
image_gradient_orientation	Calculation of image directional information () based on a discrete calculation of arc tangent.
image_edge_thinning	Merging of dual-edge borders generated by use of directional gradient masks into single-edge borders.
hough_transform_accumulate	Parametrized pattern recognition of a predefined template using the Generalized Hough Transform.
image_max_search	Search for the maximum value of an 8-bit unsigned integer image.

3 Supported Compilers

The library was built and tested using the following compilers:

- CodeWarrior — For MPC5500 V2.3
- Green Hills — MULTI for PowerPC v5.0.5

4 Directory Structure

- doc — Contains library user's manual

CodeWarrior:

- cw — Contains project files to rebuild the library
  - bin — Contains library file SPE_Image_Processing_cw.a
  - lcf — Linker command files
  - SPE_Image Processing_cw_Data — Contains project data relative to library build.
  - src — Contains library source files for the CodeWarrior compiler.
    - include\ — Contains the function definitions header file SPE_Image_Processing.h

Green Hills:

- ghs — Contains project and make files to rebuild the library
  - lib — Contains library file SPE_ Image_Processing _ghs.a
  - src — Contains library source files for Green Hills compiler
    - include — Contains the function definitions header file SPE_ Image_Processing.h

5 How to Use the Library in a Project

CodeWarrior:

- Add SPE_ Image_Processing _cw.a into the project window
- Add a path to the library include file SPE_ Image_Processing.h to Target Settings (Alt-F7)->Access Paths->User Paths
- Include file SPE_ Image_Processing.h to the source file

Green Hills:

- Use –l and –L linker options. For example, –l Image_Processing_ghs.a –L{path SPE_Image_Processing_ghs.a}
- Use –I compiler option. For example, –I{path to SPE_Image_Processing.h}
- Include file SPE_Image_Processing.h to the source file
- The –l, –L, and –I options can be set in the MULTI Project Builder menu Edit->Set Options...->Basic Options->Project

6 How to Rebuild the Library

The project files needed to rebuild the library are stored in the project directory.

- CodeWarrior — Open project SPE_Image_Processing_cw.mcp in the CodeWarrior IDE and press F7.
- Green Hills — Open project SPE_Image_Processing_ghs.gpj in the MULTI Project Builder and press Ctrl+F7.

7 Function API

7.1 Integer Spatial Filtering 2x2 (Correlation)

Function Call — void spatial_filt_int_2x2 (UINT8 *Image, UINT16 rows, UINT16 cols, float *Mask, UINT8 *OutImage)

Arguments:

Image	in	• Pointer to input image • Is rows x cols array of 8-bit unsigned integers. • Has a memory alignment of 8 bytes
rows	in	• Number of input and output images • Are a 16-bit unsigned integer
cols	in	• Columns, are a number of input and output images • Are a 16-bit unsigned integer
Mask	in	• Pointer to a 2 x 2 Mask • Mask coefficients are floating-point numbers • Usual masks are average mask, band-pass mask, and others
OutImage	out	• Pointer to output image • Is rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes

Description — Filtering of a rows x cols, Image (8-bit unsigned integers) using a 2 x 2 mask based on spatial correlation. The result is stored in the OutImage.

Algorithm:

- Set Image are a rows x cols array.
- Set Mask is a 2 x 2 filtering mask.
- OutImage is the spatial correlation (filtering) of the Image using Mask:

$$OutImage_{ij+1} = Image_{ij}Mask_{1,1} + Image_{ij+1}Mask_{1,2} + Image_{i+1j}Mask_{2,1} + Image_{i+1,j+1}Mask_{2,2} \quad \textit{Eqn. 1}$$

Equation 1 is valid for each pair of i and j with $1 \le i \le rows - 1$ and $1 \le j \le cols - 1$.

NOTE

First column and last row of output images are not calculated. These values are left unchanged. See Section 8, “Performance”.

Example 1. spatial_filt_int_2x2

```
unsigned short rows = 308;
unsigned short cols = 308;
unsigned int size = rows*cols;
unsigned char Image [size] = {0};
/* At this point, actual image is loaded into MCU memory */
unsigned char OutImage [size] = {0};

float AvgMask_2x2 [2] [2] =
{
        0.2500,          0.2500,
        0.2500,          0.2500
};

spatial_filt_int_2x2(&Image, rows, cols, *AvgMask_2x2, &OutImage);
```

(a) (b)

(a) Input image of size (rows x cold)
(b) Image filtered using an average mask

Figure 1. Effect of Applying 2x2 Spatial Filtering Based on Correlation

7.2 Integer Spatial Filtering 3x3 (Correlation)

Function Call — void spatial_filt_int_3x3(UINT8 *Image, UINT16 rows, UINT16 cold, float *Mask, UINT8 *Automate)

Arguments:

Image	in	• Pointer to input image • Is in rows x cold array of 8-bit unsigned integers • Has a memory alignment of 8 bytes
rows	in	• Number of input and output images • Are a 16-bit unsigned integer
cold	in	• Columns, number of input and output images • Are a 16-bit unsigned integer
Mask	in	• Pointer to a 3 x 3 Mask • Coefficients are floating-point numbers
Automate	out	• Pointer to output image • Are a rows x cold array of 8-bit unsigned integers • Has a memory alignment of 8 bytes

Description — Filtering of a rows x cold Image (8-bit unsigned integers) using a 3 x 3 mask based on spatial correlation. The result is stored into a rows x cold automate of 8-bit unsigned integers.

Algorithm:

- Set Image to a rows x cold array
- Set Mask to a 2 x 2 filtering mask
- Automate is the spatial correlation (filtering) of Image using Mask:

$$Automate_{i+1,j+1} = Image_{ij}Mask_{1,1} + \ldots + Image_{ij+2}Mask_{1,3} + Image_{i+1,j}Mask_{2,1} + \ldots + Image_{i+1,j+2}Mask_{2,3} + Image_{i+2,j}Mask_{3,1} + \ldots + Image_{i+2,j+2}Mask_{3,3} \quad \textit{Eqn. 2}$$

Equation 2 is valid for each pair of i and j with $1 \le i \le rows - 2$ and $1 \le j \le cols - 2$.

Applications — Examples of commonly used filtering masks are:

- Gaussian
- And
- Average
- Prewitt
- Sobel

NOTE

First and last columns, as well as first and last rows of the output image are not calculated. These values are left unchanged. See Section 8, "Performance".

Example 2. spatial_filt_int_3x3

```
unsigned short rows = 308;
unsigned short cols = 308;
unsigned int size = rows*cols;
unsigned char Image [size] = {0};
/* At this point, actual image is loaded into MCU memory */
unsigned char OutImage [size] = {0};

float GaussianMask_3x3 [3] [3] = {
    0.1088,    0.1123,    0.1088,
    0.1123,    0.1158,    0.1123,
    0.1088,    0.1123,    0.1088
};

spatial_filt_int_3x3(&Image, rows, cols, *GaussianMask_3x3, &OutImage);
```

(a) (b)

(a) Input Image of size (rows x cols)
(b) Image filtered using a Gaussian mask

Figure 2. Effect of Applying 3x3 Spatial Filtering Based on Correlation

7.3 Image Binarize

Function Call — void image_binarize(UINT8 *Image, UINT16 rows, UINT16 cols, UINT8 Threshold, UINT8 *OutImage)

Arguments:

Image	in	• Pointer to input image • Is in rows x cols array of 8-bit unsigned integers • Image has memory alignment of 8 bytes
rows	in	• Number of input and output images • Are a 16-bit unsigned integer.
cols	in	• Columns, number of input and output images • Are a 16-bit unsigned integer.
Threshold	in	• Reference value to binarize • Is an 8-bit unsigned integer
OutImage	out	• Pointer to output image • Is a rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes

Description — Compute binary representation of an 8-bit unsigned integer image using a thresholding approach. Binary representation is stored into an output image OutImage.

Algorithm:

- Set image to be a rows x cols array
- Set threshold to be a reference value
- Then, OutImage is the binary representation of Image respect given by:

$$OutImage_{i,j} = \begin{cases} 0 Image_{i,j} \le Threshold \\ 255 Image > Threshold \end{cases} \quad \textit{Eqn. 3}$$

Equation 3 is valid for each pair of i and j with $1 \le i \le rows$ and $1 \le j \le cols$.

Applications — By modifying threshold, this function can erase undesired information (noise) of an image.

NOTE

See Section 8, “Performance”.

Example 3. image_binarize

```
unsigned short rows = 308;
unsigned short cols = 308;
unsigned int size = rows*cols;
unsigned char Image [size] = {0};
/* At this point, actual image is loaded into MCU memory */
unsigned char OutImage [size] = {0};

image_binarize(&Image, rows, cols, 120,&OutImage);
```

(a) (b)

(a) is the input image without processing
(b) is the output image after processing

Figure 3. Input and Output Images

7.4 Image Subtraction

Function Call — void image_subtract(UINT8 *ImageA, UINT8 *ImageB, UINT16 rows, UINT16 cols, UINT8 *OutImage)

Arguments:

ImageA	in	• Pointer to input image • Is a rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes
ImageB	in	• Pointer to input image • Is a rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes
rows	in	• Number of inputs and output images • Are a 16-bit unsigned integer
cols	in	• Columns, number of inputs and output images • Are a 16-bit unsigned integer
OutImage	out	• Pointer to output image • Is a rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes

Description — Subtraction of 8-bit unsigned integer images, ImageA and ImageB. The result is stored into an output image, OutImage of 8-bit unsigned integers.

Algorithm — If ImageA and ImageB are arrays of identical dimensions rows x cols, their subtraction is then a rows x cols array denoted by OutImage = ImageA – ImageB. See Equation 4

$$OutImage_{i,j} = ImageA_{ij} - ImageB_{i,j} \qquad \textit{Eqn. 4}$$

Equation 4 is valid for each pair of i and j with $1 \le i \le rows$ and $1 \le j \le cols$.

Applications — This function can highlight features that differ from one image to another. For example, a Gaussian filtered version of an image can be subtracted to perform an unsharp masking operation.

Unsharp Masking Approach Using Image Subtraction.

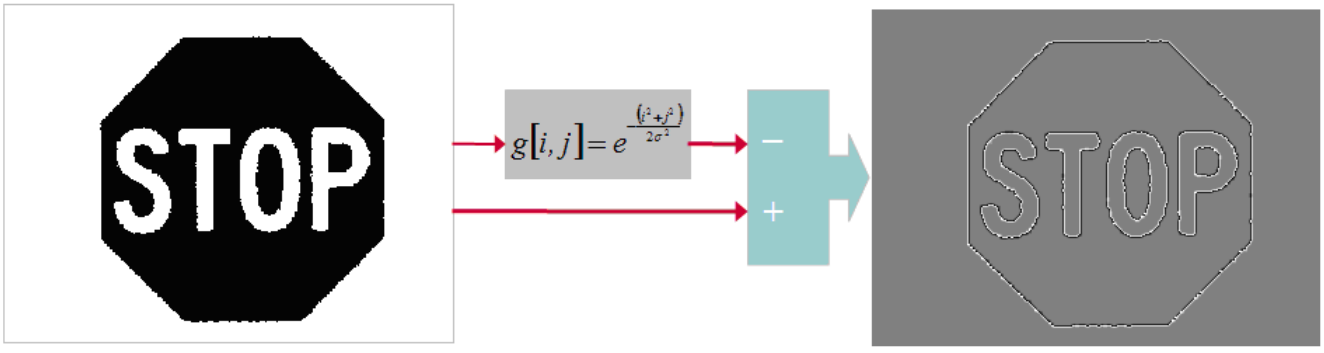


Figure 4. Unsharp Masking

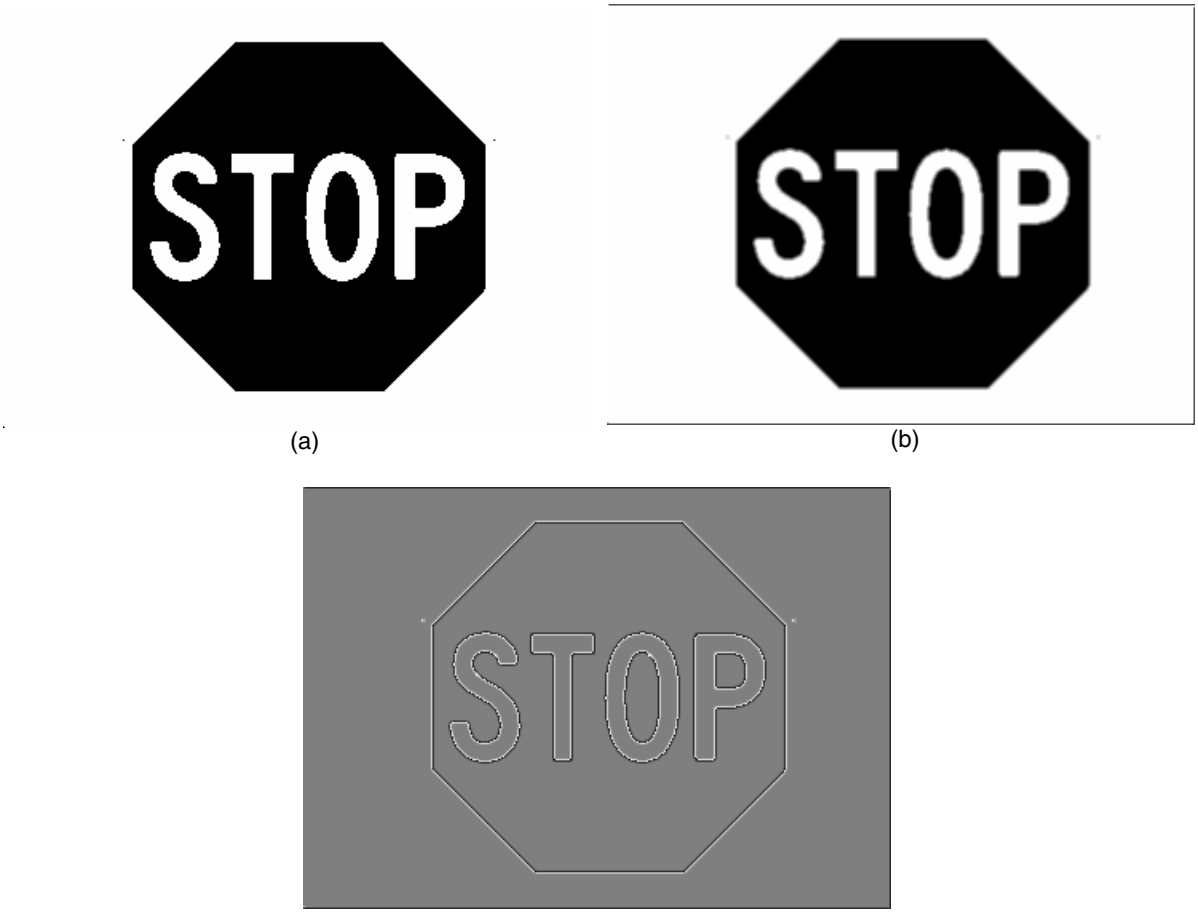
NOTE

See Section 8, “Performance”.

Example 4. image_subtract

```
unsigned short rows = 308;
unsigned short cols = 308;
unsigned int size = rows*cols;
unsigned char ImageA [size] = {0};
unsigned char ImageB [size] = {0};
/* At this point, actual images are loaded into MCU memory */
unsigned char OutImage [size] = {0};

image_subtract(&ImageA, &ImageB, rows, cols, &OutImage);
```



(a) (b)

(c)

(a) and (b) are the input images
(c) is the output image

Figure 5. Input and Output Images for Subtraction

7.5 Image Gradient Orientation

Function Call — void image_gradient_orientation(INT8 *GradY, INT8 *GradX, UINT8 *PhiImage,UINT32 ImageSize, INT16 *ArcTanTable)

Arguments:

GradY	in	• Pointer to input image • Is a rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes • Contains $\frac{\partial A(x,y)}{\partial y}$ information of Image A
GradX	in	• Pointer to input image • Is a rows x cols array of 8-bit unsigned integers. • Has a memory alignment of 8 bytes • Contains $\frac{\partial A(x,y)}{\partial x}$ information of Image A
PhiImage	out	• Pointer to output image • Is a rows x cols image of 8-bit unsigned integers • Has a memory alignment of 8 bytes
ImageSize	in	• Number of pixels of input and or output images (rows x cols) • Is a 32-bit unsigned integer.
ArcTanTable	in	• Discrete values of Arc tan function as Table 1 • Are a 16-bit signed integer.

Description — Compute directional information () based on a discrete calculation of the Arc tangent. The resulting gradient orientation image is stored into PhiImage.

Algorithm:

The directional information of an image is:

$$\theta(x,y) = arc\tan\left[\frac{G_y}{G_x}\right] \qquad \textit{Eqn. 5}$$

$Grad_y$ and $Grad_x$ are:

$$\begin{bmatrix} Grad_x \\ Gfad_y \end{bmatrix} = \begin{bmatrix} \frac{\partial A(x,y)}{\partial x} \\ \frac{\partial A(x,y)}{\partial y} \end{bmatrix} \qquad \textit{Eqn. 6}$$

Then,

$$\tan(\theta(x,y)) = \frac{Grad_y}{Grad_x} \qquad \textit{Eqn. 7}$$

Table 1 shows discrete values of tan θ in discrete (scaled) increments of 10°.

Table 1. Discreet Values for tan θ

Degrees	Radians	tanθ	Discrete value
95	1.658062789	–11.430052	–732
105	1.832595715	–3.732051	–239
115	2.00712864	–2.144507	–137
125	2.181661565	–1.428148	–91
135	2.35619449	–1.000000	–64
145	2.530727415	–0.700208	–45
155	2.705260341	–0.466308	–30
165	2.879793266	–0.267949	–17
175	3.054326191	–0.087489	–6
5	0.087266463	0.087489	6
15	0.261799388	0.267949	17
25	0.436332313	0.466308	30
35	0.610865238	0.700208	45
45	0.785398163	1.000000	64
55	0.959931089	1.428148	91
65	1.134464014	2.144507	137
75	1.308996939	3.732051	239
85	1.483529864	11.430052	732

NOTE

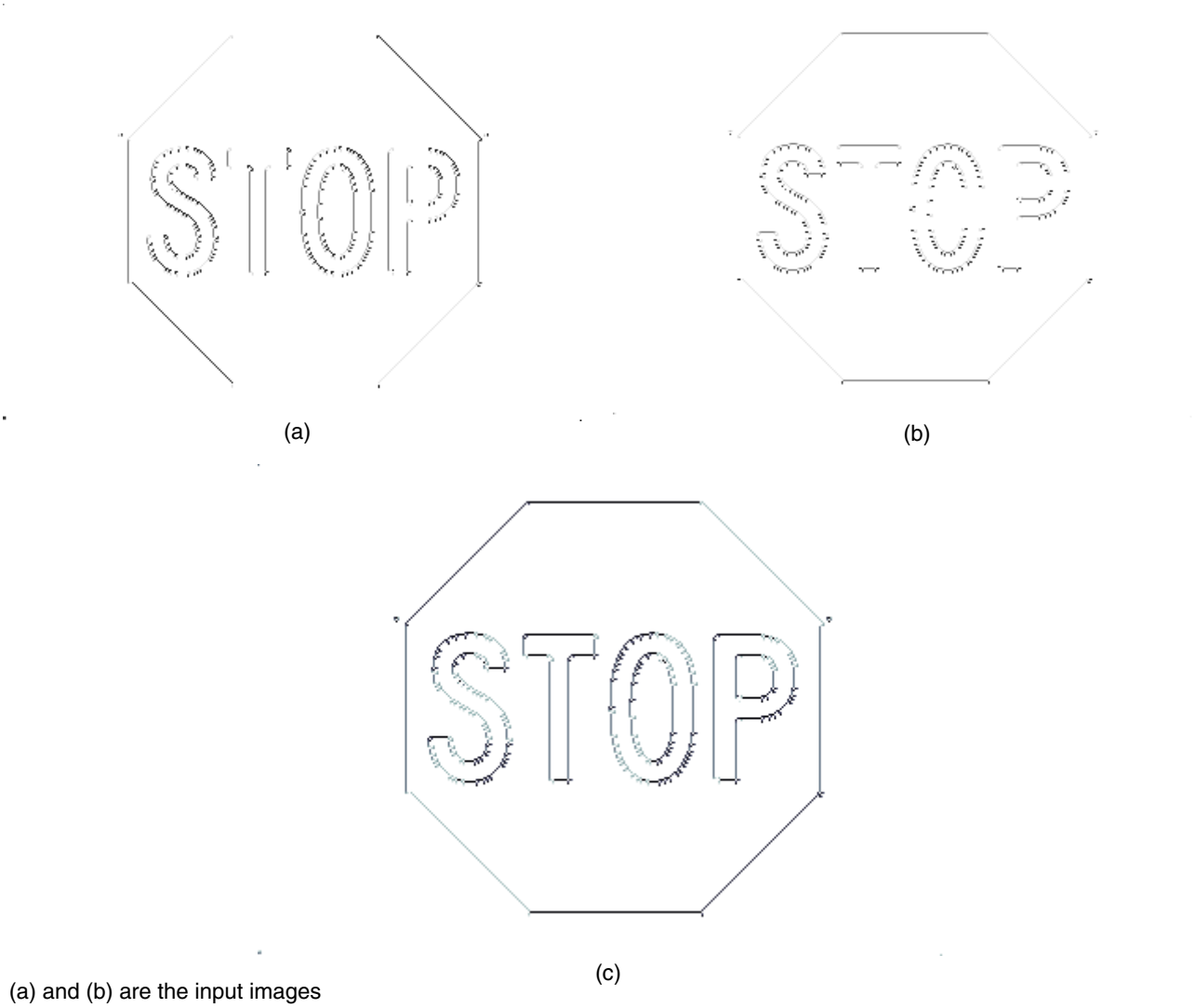
See Section 8, “Performance”.

Example 5. image_gradient_orientation

```
unsigned short rows = 264;
unsigned short cols = 368;
unsigned int size = rows*cols;
unsigned char GradX [size] = {0};
unsigned char GradY [size] = {0};
/* At this point, actual images are loaded into MCU memory */
unsigned char PhiImage [size] = {0};

short   ArcTanTable[18]= {-732,-239,-137,-91,-64,-45,-30,-17,-6,
6,17,30,45,64,91,137,239,732};

image_gradient_orientation(&GradY, &GradX, &PhiImage, size, ArcTanTable);
```



(a) and (b) are the input images
(c) is the output image

Figure 6. Input and Output Images of Image Gradient Orientation

7.6 Edge Thinning

Function Call — void image_edge_thinning (INT8 *GradY, INT8 *GradX, UINT16 rows, UINT16 cols).

Arguments:

GradY	in/out	• Pointer to input image • Are a rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes • Contains $\frac{\partial A(x,y)}{\partial y}$ information of Image A
GradX	in/out	• Pointer to input image • Are a rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes • Contains $\frac{\partial A(x,y)}{\partial x}$ information of Image A
rows	in	• Number of input/output images • Are a 16-bit unsigned integer
cols	in	• Columns, number of input/output images • Are a 16-bit unsigned integer

Description — This function merges dual-edge borders generated by using directional gradient masks, as in Ando's[1] directional masks into single-edge borders. The resulting single-edge border images are stored into the same input images GradY and GradX.

Algorithm — Dual-edge border merging into single-edge border images is performed using the following approach:

Define the following sets. These are based on actual pixel content.

$$
\begin{aligned}
same(i,j) &= \{GradX(i,j)GradY(i,j) | GradX(i,j) = GradY(i,j)\} \\
sum(i,j) &= \{GradX(i,j), GradY(i,j) | GradX(i,j) + GradY(i,j) = GradY(i,j)\} \\
A(i,j) &= \{GradX(i,j) | 35 < GradX(i,j) \le 125\} \\
B(i,j) &= \{GradY(i,j) | 35 < GradY(i,j) \le 125\} \\
C(i,j) &= \{GradX(i,j) | 128 < GradX(i,j) \le 220\} \\
D(i,j) &= \{GradY(i,j) | 128 < GradY(i,j) \le 220\}
\end{aligned}
$$

Eqn. 8

1. S. Ando, "Consistent Gradient Operators", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 22, No. 3, pp. 252-255, March 2000

Next, define its corresponding indexes as:

$$
\begin{aligned}
(i,j)_a &= \{(i,j) \in A(i,j) \cap B(i,j)\} \\
(i,j)_b &= \{(i,j) \in A(i,j) \cap D(i,j)\} \\
(i,j)_c &= \{(i,j) \in C(i,j) \cap B(i,j)\} \\
(i,j)_d &= \{(i,j) \in C(i,j) \cap D(i,j)\} \\
(i,j)_e &= \{(i,j) \in same(i,j) \cap (i,j)GradY((i,j) > 69)\} \\
(i,j)_f &= \{(i,j) \in sum(i,j) \cap (i,j)GradY((i,j) > 186)\}
\end{aligned}
\qquad \textit{Eqn. 9}
$$

Then, define set of indexes to be zeroed-out as:

$$
(i,j)_{zero} = \{(i,j)_a \cup (i,j)_b \cup (i,j)_c \cup (i,j)_d \cup (i,j)_e \cup (i,j)_f\} \qquad \textit{Eqn. 10}
$$

Finally, single-edge border images are given by:

$$
\begin{aligned}
GradY(i,j) &= \{0, GradY(i,j) | GradY(i,j) \in (i,j)_{zero}\} \\
GradX(i,j) &= \{0, GradX(i,j) | GradX(i,j) \in (i,j)_{zero}\}
\end{aligned}
\qquad \textit{Eqn. 11}
$$

Equation X is valid for each pair of i and j with $1 \le i \le rows$ and $1 \le j \le cols$.

Applications — After using Ando's directional gradient masks to obtain GradX and GradY, this function merges dual-edge border into single-edge border images. These resulting images can be used in a Hough Transform algorithm reducing in half the number of border information to process.

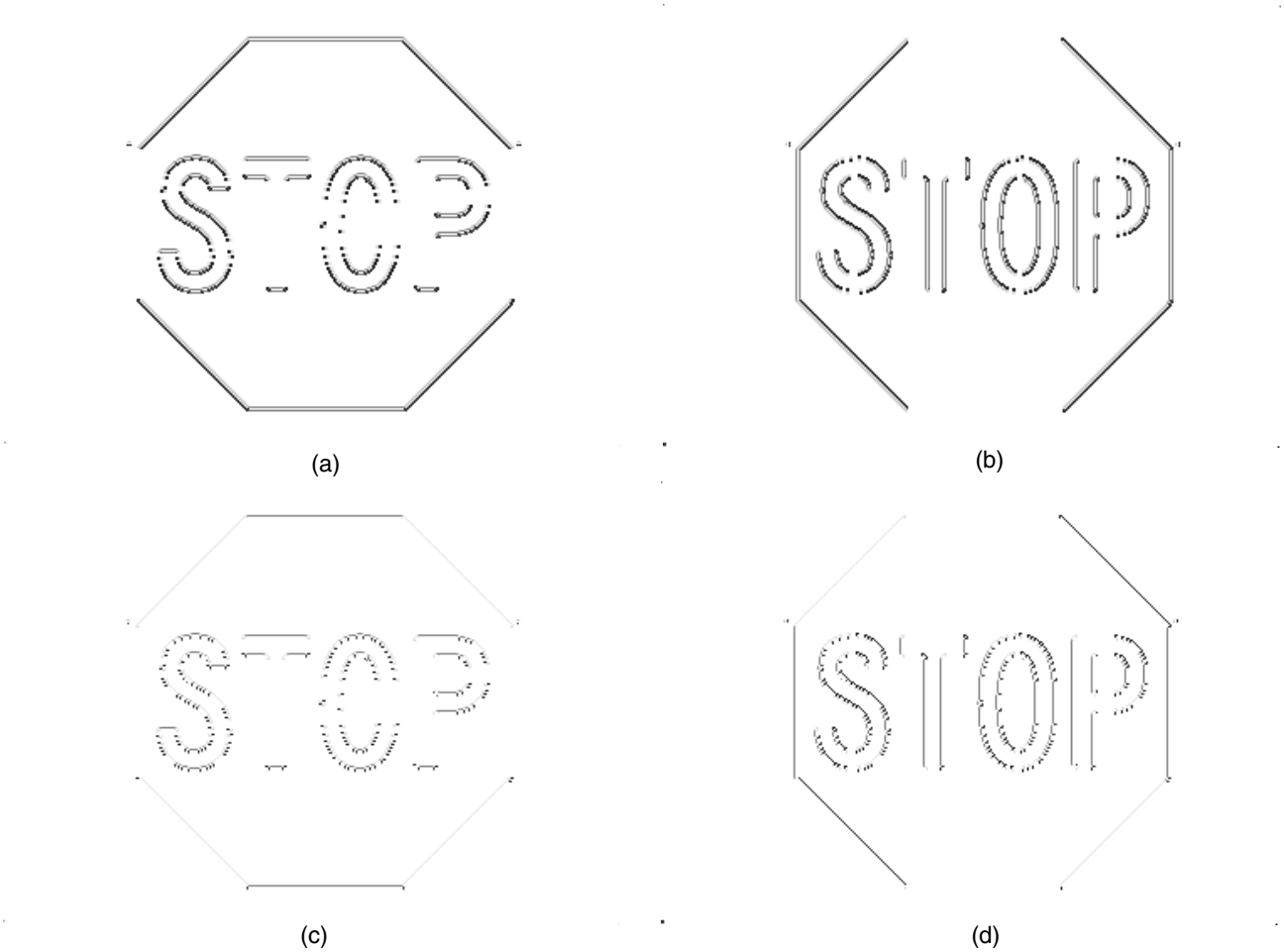
NOTE

See Section 8, "Performance".

Example 6. image_edge_thinning

```
unsigned short rows = 264;
unsigned short cols = 368;
unsigned int size = rows*cols;
unsigned char GradX [size] = {0};
unsigned char GradY [size] = {0};
/* At this point, actual images are loaded into MCU memory */

image_edge_thinning(*GradY, *GradX, rows, cols);
```



(a) and (b) are the input images
(c) and (d) are the output images

Figure 7. Input and Output Images of Image Thinning

7.7 Generalized Hough Transform

Function Call — void hough_transform_accumulate (UINT8 *ThetaImage,UINT16 rows,UINT16 cols, UINT16 *ThetaLength,UINT16 *ThetaIndex, INT16 *AlphaSine,INT16 *AlphaCos,INT16 *R,UINT8 *HoughAcc)

Arguments:

ThetaImage	in	• Pointer to input image • Is is a rows x cols array of 8-bit unsigned integers • Has a memory alignment of 8 bytes • Contains image gradient orientation resulting from image_gradient_orientation or image_edge_thinning function calls
rows	in	• Number of input and output images • 16-bit unsigned integer
cols	in	• Number of input and output images • 16-bit unsigned integer
ThetaLength	in	• Pointer to the number of elements with the same directional orientation • 16-bit unsigned integer • Has a memory alignment of 4 bytes
ThetaIndex	in	• Pointer to localize elements with the same directional orientation • 16-bit unsigned integer • Has a memory alignment of 4 bytes
AlphaSine	in	• Pointer to sine function of local edge direction angle • 16-bit signed integer • Has a memory alignment of 4 bytes
AlphaCos	in	• Pointer to cosine function of local edge direction angle • Is a 16-bit signed integer • Has a memory alignment of 4 bytes
R	in	• Pointer to R-table • 16-bit signed integer • Contains the table of Euclidean distance between the template boundary points and an arbitrary reference point • Has a memory alignment of 4 bytes
HoughAcc	Out	• Pointer to discrete Hough domain output image, or Hough accumulator • 8-bit unsigned integer • Has a memory alignment of 8 bytes

Description — This method requires a template image and uses the directional information available. Each boundary point $P_i(x_i, y_i)$ is represented as an $r\alpha$ pair. In Figure 8, r is the Euclidean distance from a reference point $P_i(x_i, y_i)$ to the boundary point, and α is the angle of the line connecting the boundary point and the reference point.

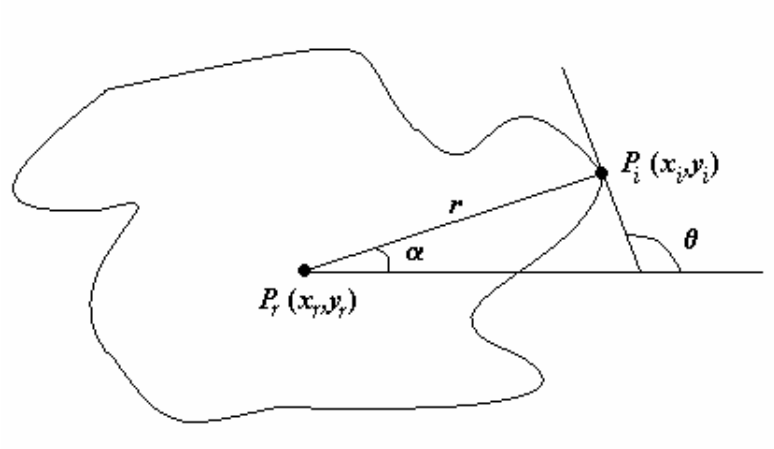


Figure 8. Image Information

These pairs are stored into a list called the R-table, indexed by the local edge direction θ_i at the boundary point. In general, mapping the R-table represents a many-to-one mapping, empty entries are allowed. Table 2 shows the R-table.

Table 2. R-Table

θ_i	(r,α)
$\Delta\theta$	(r_1,α), (r,α_1)
$2\Delta\theta$	(r_3,α_3)
$3\Delta\theta$	(r_4,α_4), (r_5,α_5), (r_6,α_6)
...	...

For this function *r* information is called from the parameter R.

Alpha (α) information must be stored in AlphaSine and AlphaCos in the form of $\sin(\alpha)$ and $\cos(\alpha)$ multiplied by a factor of 256. Theta (θ) information must be stored in ThetaLength and ThetaIndex. It must contain the number of elements with the same orientation and their absolute index within the template images.

Algorithm — Figure 9 shows the algorithm for this function.

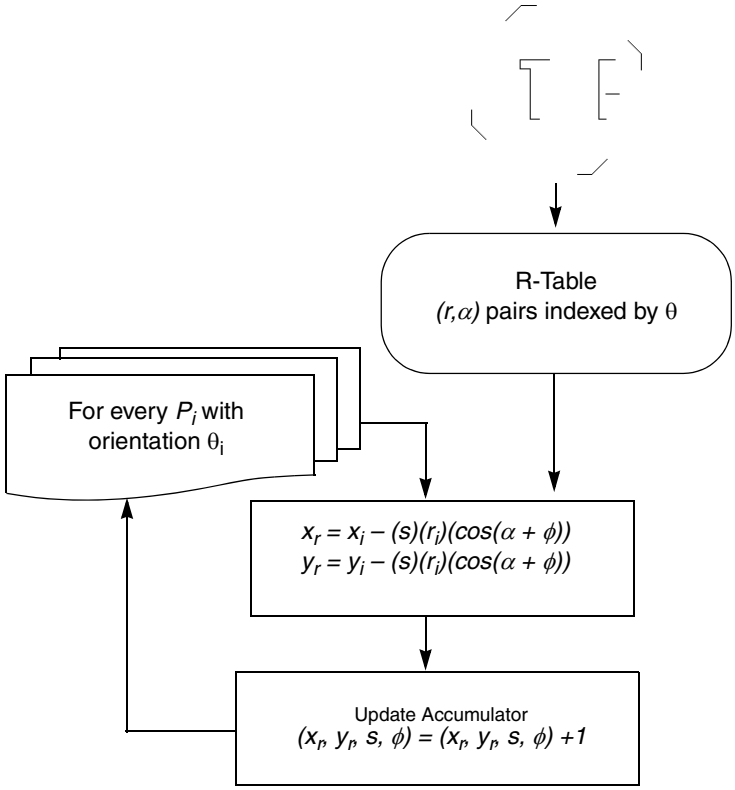


Figure 9. Algorithm

Applications — The user can take advantage of the Hough transform function to implement template segmentation techniques that have little variance to rotation, scale, noise, occlusions, and shape variability. One application is recognition of traffic signs along the road by an on-board vehicle computer paired with an imaging sensor.

NOTE

See Section 8, “Performance”.

Example 7. hough_transform_accumulate

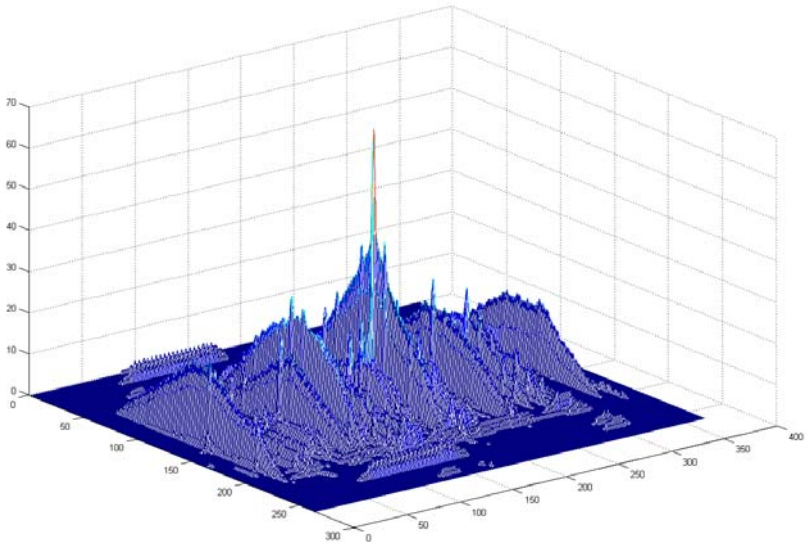
```
unsigned short rows = 264;
unsigned short cols = 368;
unsigned int size = rows*cols;
unsigned char ThetaImage[size] = {0};
/* At this point, actual image is loaded into MCU memory */
unsigned char HoughAcc[size] = {0};

/*
ThetaImage, Temp_STOP_theta_lenght, Temp_STOP_theta_index, Temp_STOP_sine_alpha,
Temp_STOP_cosine_alpha and Temp_STOP_r parameters are initialized with the information of
template shown in Figure 10 (b). (See section “Appendix A, Template Definition for Hough
Transform applications”)
*/
```

```
hough_transform_accumulate(&ThetaImage, rows, cols, &Temp_STOP_theta_lenght[0][0],
&Temp_STOP_theta_index[0][0], &Temp_STOP_sine_alpha[0][0], &Temp_STOP_cosine_alpha[0][0],
&Temp_STOP_r[u8Index][0], &HoughAcc);
```

(a)

(b)



(c)

(a) Image to be processed
(b) Template
(c) Resulting accumulator after Hough transformation

Figure 10. Hough Transform Accumulator

7.8 Image Maximum Search

Function Call — void image_max_search(UINT8 *Image ,UINT16 rows, UINT16 cols, UINT8 *MaxValue,UINT16 *MaxIndex_x, UINT16 *MaxIndex_y);

Arguments:

Image	in	• Pointer to input image • Is a rows x cols array of 8-bit unsigned integers • Image has a memory alignment of 8 bytes
rows	in	• Number of input and output images • Are a 16-bit unsigned integer
cols	in	• Number of input and output images • Are a 16-bit unsigned integer
MaxValue	out	• Maximum value found • Is an 8-bit unsigned integer
MaxIndex_x	out	• Maximum value x-coordinate • Is a 16-bit unsigned integer
MaxIndex_y	out	• Maximum value y-coordinate • Is a 16-bit unsigned integer

Description — Search for the maximum value of an 8-bit unsigned integer image of size rows x cols. Maximum value found is stored into MaxValue, and its coordinates are MaxIndex_x and MaxIndex_y.

Algorithm — Maximum value within image is defined as:

$$MaxValue(i,j) = max\{Image_{i,j}\} \qquad \textbf{Eqn. 12}$$

Equation 12 is valid for each pair of i and j with $1 \le i \le rows$ and $1 \le j \le cols$.

Applications — This function searches for the maximum value within a bi-dimensional array (image) and provides the user with the maximum found value and its x and y coordinates. An application of this function is to search for the maximum number of votes on the discrete Hough domain (accumulator) and determine its x and y coordinates within this domain.

NOTE

See Section 8, “Performance”.

Example 8. image_max_search

```
unsigned short rows = 264;
unsigned short cols = 368;
unsigned int size = rows*cols;
unsigned short MaxIndex_x = 0;
unsigned short MaxIndex_y = 0;
unsigned char MaxValue=0;
unsigned char Image[size];
/* At this point, actual image is loaded into MCU memory */

image_max_search(*Image , rows, cols, *MaxValue, *MaxIndex_x, *MaxIndex_y);
```

image_max_search function call yields the following results when Image shown in Figure 10 (c) is processed:

```
MaxValue = 67
MaxIndex_x = 190
MaxIndex_y = 131
```

8 Performance

Figure 11. Code Size per Function

	Function Name	Code Size (Bytes)
1	spatial_filtering_int_2x2	316
2	spatial_filtering_int_3x3	732
3	image_binarize	300
4	image_subtract	332
5	image_gradient_orientation	704
6	image_edge_thinning	448
7	hough_transform_accumulate	540
8	image_max_search	268

Table 3. Execution Time[1] Relative to Pixels

	Function Name	Execution Time (Clock Cycles/pixel)
1	spatial_filtering_int_2x2	13.8
2	spatial_filtering_int_3x3	20.7
3	image_binarize	5.8
4	image_subtract	3.8
5	image_gradient_orientation*	8.7
6	image_edge_thinning*	8.9
7	hough_transform_accumulate*	51
8	image_max_search	9.8

1 *Performance of functions noted with an asterisk (*) and vary depending upon image content. The values shown in Table 3 represent average performance obtained during development and or testing of a limited number of images.

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

AN3806
Rev. 0
02/2009