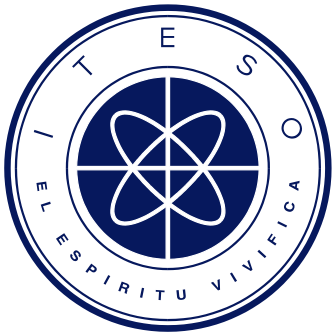
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS



BIBLIOTECA DE PROCESAMIENTO DE IMÁGENES OPTIMIZADA PARA ARM® CORTEX®-M7

Tesina para obtener el grado de:

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: Roberto Ortega Hernández

Director: Abraham Tézmol Otero

San Pedro Tlaquepaque, Jalisco. Julio de 2017.

DEDICATORIA

A mi familia, especialmente a mis padres Leticia y Armando, por estar siempre a mi lado, por ofrecerme los medios para superarme profesionalmente y por su incansable apoyo para cada uno de mis proyectos, por esto y más les estaré eternamente agradecido.

A mi esposa Mónica, mi mayor inspiración, por alimentar mis deseos de superación profesional con su incesante espíritu de aprendizaje y por brindarme su amor incondicional día a día durante esta aventura.

AGRADECIMIENTOS

Quiero agradecer al Consejo Nacional de Ciencia y Tecnología (Conacyt) por fomentar la profesionalización de los ciudadanos y el desarrollo científico y tecnológico de México.

Agradezco también a la empresa Continental por su alto compromiso con el desarrollo de sus empleados y por las facilidades otorgadas para llevar a cabo este programa de especialidad.

Por último, quiero agradecer al profesor Abraham Tézmol por su destacable dedicación a la enseñanza y por el apoyo brindado durante la realización de este proyecto.

Abstract

Most modern vehicles are equipped with systems that assist the driver by automating difficult and repetitive tasks, such as reducing the vehicle speed in a school zone. Some of these systems require an onboard computer capable of performing real-time processing of the road images captured by a camera. The goal of this project is to implement an optimized image processing library for the ARM® Cortex®-M7 architecture. This library includes the routines to perform image spatial filtering, subtraction, binarization, and extraction of the directional information along with the parameterized pattern recognition of a predefined template using the Generalized Hough Transform (GHT). These routines are written in the C programming language, leveraging GNU ARM C compiler optimizations to obtain maximum performance and minimum object size. The performance of the routines was benchmarked with an existing implementation for a different microcontroller, the Freescale® MPC5561. To prove the usability of this library in a real-time application, a Traffic Sign Recognition (TSR) system was implemented. The results show that in average the execution time is 18% faster and the binary object size is 25% smaller than in the reference implementation, enabling the TSR application to process up to 24 fps. In conclusion, these results demonstrate that the image processing library implemented in this project is suitable for real-time applications.

Resumen

La mayoría de los vehículos en la actualidad están equipados con sistemas que asisten al conductor en tareas difíciles y repetitivas, como reducir la velocidad del vehículo en una zona escolar. Algunos de estos sistemas requieren una computadora a bordo capaz de realizar el procesamiento en tiempo real de las imágenes del camino obtenidas por una cámara. El objetivo de este proyecto es implementar una librería de procesamiento de imagen optimizada para la arquitectura ARM® Cortex®-M7. Esta librería provee rutinas para realizar filtrado espacial, resta, binarización y extracción de la información direccional de una imagen, así como el reconocimiento parametrizado de patrones de una figura predefinida utilizando la Transformada Generalizada de Hough. Estas rutinas están escritas en el lenguaje de programación C, aprovechando las optimizaciones del compilador GNU ARM C, para obtener el máximo desempeño y el mínimo tamaño de objetos. El desempeño de las rutinas fue comparado con la implementación existente para otro microcontrolador, el Freescale® MPC5561. Para probar la funcionalidad de esta librería en una aplicación de tiempo real, se desarrolló un sistema de reconocimiento de señales de tráfico. Los resultados muestran que en promedio el tiempo de ejecución es 18% más rápido y el tamaño de objetos es 25% menor que en la implementación de referencia, lo que habilita a este sistema para procesar hasta 24 cuadros por segundo. En conclusión, estos resultados demuestran la funcionalidad de la librería de procesamiento de imágenes en sistemas de tiempo real.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ADAS.....Advanced Driver Assistance Systems
AHB .....Advanced High-performance Bus
ARM .....Advanced RISC Machine
AXI .....Advanced Extensible Interface
CPU.....Central Processing Unit
DMA .....Direct Memory Access
DSP .....Digital Signal Processor
DTCM.....Data Tightly Coupled Memory
DWT .....Data Watchpoint and Trace
ECC.....Error-correcting Code
ETM .....Embedded Trace Macrocell
FPGA .....Field-programmable Gate Arrays
fps.....Frames per second
FPU .....Floating Point Unit
GHT .....Generalized Hough Transform
GNU.....GNU's not Unix
ISA .....Instruction set architecture
ITCM.....Instruction Tightly Coupled Memory
ITESO .....Western Institute of Technology and Higher Education
MAC .....Multiply–accumulate
MCU .....Microcontroller Unit
ms.....millisecond
NVM .....Non-volatile Memory
PowerPC .....Performance Optimization with Enhanced RISC Performance Computing
RISC.....Reduced Instruction Set Computer
RAM .....Random Access Memory
RWW .....Read While Write
SRAM .....Static Random Access Memory
SIMD.....Single instruction, multiple data
TCM.....Tightly Coupled Memories
TSR .....Traffic Sign Recognition
VFP .....Vector Floating-Point

Introduction

Advanced Driver Assistance Systems (ADAS) are systems that help motorists while driving. In general, these systems increase automobile and road safety by means of providing the driver, or other driving assistance systems, with real-time information via an appropriate interface while automating difficult and repetitive tasks, such as reducing the speed to avoid a collision [1]. ADAS have recently become part of most modern automobiles, including commercial vehicles. Safety is the main incentive for the growth of ADAS since the number of motor vehicle collisions is high enough to be considered an epidemic, while driver comfort is also a key motivator [2].

Some examples of ADAS are lane, vehicle, and pedestrian detection, forward collision warning, and parking assistance, among others. Several of these technologies require image processing capabilities of the vehicle's onboard computer, such as Traffic Sign Recognition (TSR).

TSR is used to regulate traffic signs, warn drivers, and command or prohibit certain actions such as limiting or reducing the vehicle speed in a school zone. By liberating the driver of these tasks, TSR increases driving safety and comfort [3]. A key aspect of TSR is a fast and robust embedded system capable of sensing the environment and analyzing the data to detect traffic signs. To do this, an image sensor is used to capture a picture of the road, then this picture is analyzed using different detection methods such as color or shape segmentation. The final stage is to present the information to the driver or send it to other automobile subsystems in order to act accordingly (e.g. drive the automobile in the correct lane).

The vehicle´s onboard computer must support processor optimized software routines for image processing when implementing TSR. These routines must perform image filtering and segmentation to recognize patterns. Given that TSR is required to provide real-time information, these routines must complete the video frames analysis fast enough so that these are still up to date [4].

For this reason, the goal of this project is to implement an optimized software library for the ARM® Cortex®-M7 architecture that contains the image processing routines needed to perform shape recognition, using Generalized Hough Transform (GHT). These routines include image spatial filtering, subtraction, binarization, and extraction of the directional information along with the parameterized pattern recognition of a predefined template using the GHT. The results will be benchmarked with the existing implementation for a different microcontroller (Freescale® MPC5561) to prove its usability in a real-time application.

The processor used for this project is the Atmel® SAMV71Q21. It is based on the 32-bit ARM® Cortex®-M7 architecture. This processor is present in the Atmel® SAM V71 Xplained Ultra evaluation kits, which have recently been acquired by the Western Institute of Technology and Higher Education (ITESO) for its Embedded Systems Specialization Program. By using this evaluation kit, future students will be empowered to develop other interesting projects using the image processing library implemented for this project as a building block of any system.

1. Background

Nowadays, image processing is an important area of research mainly because of its wide range of applications. Image processing is applied in different fields such as industrial automation [5], automotive industry [6], and the medical sector [7]. Sophisticated image processing techniques are also used to diagnose deviations of the spine in a patient the same way they help identify flaws in a product assembly line. Modern automobiles can identify traffic signs by means of applying similar image processing techniques.

One of the most habitual problems in image processing applications is the existence of noise in the images that are processed [5]. Noise can be introduced during the image acquisition, transmission, and compression phases [8]. This undesired information must be removed from the image before further processing; this can be achieved using a digital signal processing technique known as spatial filtering. In this technique, each pixel of the image is modified by its neighborhood and a matrix of coefficients named mask. By using this method, several other image effects can be achieved just by changing the coefficients of the filter's mask, such as smoothing, edge detection, noise reduction, and sharpening.

The median filter is a type of spatial filter used to remove noise from an image; it is usually found in a previous stage of other image processing techniques, such as image segmentation. In an image, the median filter uses a mask that is applied to each pixel with the purpose of determining the median value. This resulting value is placed in the same pixel position of the output image [9].

Pattern recognition applications require a method that can identify shapes regardless of the noise of the image and other visual obstacles. The Generalized Hough Transform is a suitable method for this purpose [10]. GHT is a technique that searches for occurrences of a template in an image, and when a match is found, a matrix called accumulator is updated. Finally, the matrix coordinates with the higher number of votes represent the template's reference point if the number of votes is above a given threshold [7]. Fixed orientation shapes can be identified using a two-

dimensional array containing only the coordinates of the shape´s reference point. If the shape has an arbitrary orientation and scale, these values are added to the shape description array resulting in four dimensions. The disadvantage of GHT is that its algorithm has a high computational demand due to the series of nested loops resulting from its implementation [11].

Performance is important in image processing because most of the time real-time results are required (i.e. processing 30 video frames per second). Given that most of these techniques have high computational complexity due to the multidimensional nature of the signals applied [11], implementing image processing applications becomes a challenge, especially in embedded systems, where the computing power and memory bandwidth are limited.

Moreover, due to the high computational costs of spatial filtering, current research focuses on increasing the performance of the filter's core operations so that these techniques can be used in any system that requires real-time image processing. Optimizations for median filters with 3 x 3 mask using Field-programmable Gate Arrays (FPGA) have been developed, and results show that this implementation can run 85 times faster than a PC microprocessor. This optimization process produces 30 images of 640 x 480 pixels in less than a second, enabling this implementation for real-time applications [5]. Another generic solution focuses on reducing the delay time between image acquisition and image processing by means of using a line memory instead of a frame, achieving a time of 13.1 ms for filtering a frame of 640 x 512 pixels [6].

In the case of GHT, recent studies have focused on hardware implementations to achieve real-time performance. FPGA is used along with algorithm optimizations such as image downscaling and rotation to reduce the computational cost of this process [12]. Other implementations take advantage of the fact that depending on the application, only two of the four dimensions of the shape's representation are relevant, reducing the complexity by using fixed orientation match templates with no rotation or scale values [13]. GHT implementations that support the four dimensions have also been employed using a configurable FPGA architecture. Results have shown that the implementation of image processing techniques to perform shape

recognition using FPGA can result in a speed gain up to 4.5 times when compared to a high-end PC microprocessor [11].

On the other hand, FPGA disadvantages include a higher power consumption compared to microcontrollers (considering similar applications), also the set of tools and skills are very different compared to the ones used for microcontrollers. Besides, FPGA uses hardware description instead of a programming language. This increases the level of complexity of these systems along with the required expertise of the developers. Cost is also a disadvantage; FPGA chips are more expensive than microcontrollers. For this reason, having an optimized image processing library integrated into the embedded system's microprocessor represents an economic advantage compared to an FPGA dedicated image processing module.

2. Conceptual Framework

2.1. Generalized Hough Transform

The Hough Transform is an image processing technique that is useful for identifying figures that can be represented by an analytic function, such as straight lines, circles, or ellipses by operating in the edge information of an image. The number of pixels in an image that fit in an analytical figure is held in a matrix called accumulator. At the end of the process, the accumulator stores the information of the strongest analytical figure in an image [8].

The GHT was developed with the goal of identifying arbitrary shapes in an image, particularly, those shapes that cannot be represented analytically. This method requires obtaining the directional information of the image. Each boundary point $P_i(x_i,y_i)$ will be represented as a (r,α) pair. r is the Euclidean distance from a reference point $P_r(x_r,y_r)$ to the boundary point, and α is the angle of the line connecting the boundary point and the reference point as shown in Fig. 2-1.

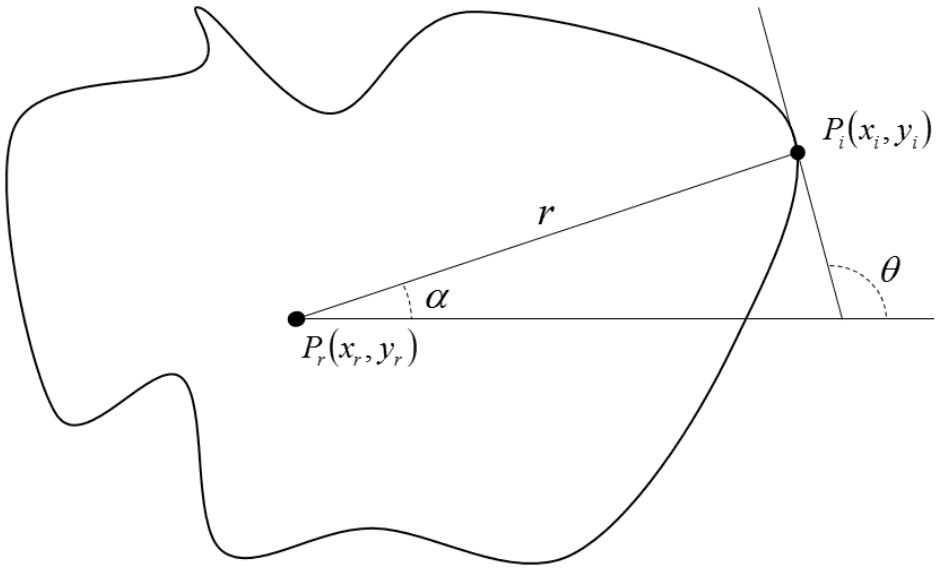


Fig. 2-1 Directional information in an image [14].

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, the mapping that the R-table represents is a many-to-one mapping where empty entries are allowed. Table I shows the form of the R-table. The parametric representation of the image is stored in the R-table in four dimensions, x_r, y_r, scale (s) and orientation (ϕ). This four-dimensional space is called Hough space.

Table I
R-table

Edge Direction θ_i	Boundary Points (r,α)
$\Delta\theta$	(r_1,α_1), (r_2,α_2)
$2\Delta\theta$	(r_3,α_3)
$3\Delta\theta$	(r_4,α_4), (r_5,α_5),(r_6,α_6)
...	...

The discrete representation of the R-table is called accumulator, where every edge point provides a vote to the corresponding Hough space bin. This is, the $(x_r,y_r,,s,\phi)$ bin in the accumulator is increased by one when a match for the reference point is found. This process is repeated for every value in the Hough space and then, the bin with the largest number of votes provides the reference point of the template in the target image [7].

2.2. Obtaining an Edge Image

Given that GHT uses the borders of an image to match the points of a template image using gradient information, it is important to have a good border image. The unsharp masking technique is used for this purpose.

Unsharp masking consists in subtracting a blurred version of an image from the original image. The blurred version of the image is obtained using a Gaussian spatial filter. Equation (2-1) defines the Gaussian filter function.

$$g[i, j] = e^{\frac{-\left(i^2 + j^2\right)}{2\sigma^2}} \tag{2-1}$$

The blurring effect of the Gaussian filter is the same in all directions and it can be controlled by σ, the greater this value is, the smoother the image will look.

The result of the blurred image subtraction from the original image is an image that contains only important variations while eliminating the unimportant information (i.e. the image's background). Fig. 2-2 shows the unsharp masking technique.

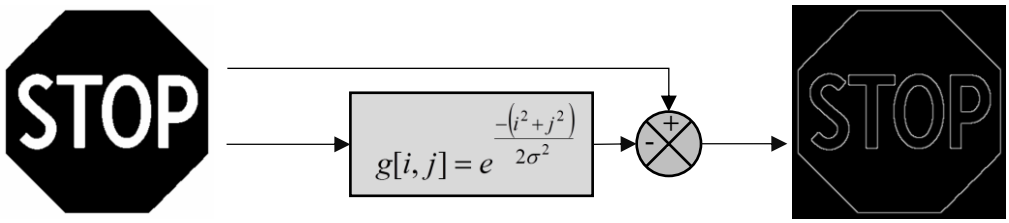


Fig. 2-2 Obtaining the edges of a sample image using unsharp masking. The blurred image (Gaussian mask) is subtracted from the original image. The resulting image contains the edges of the image [14].

2.2.1 Average Filter

The average filter is used to remove noise in an image, in this case, the edge image. This undesired information is presented as pixels changing rapidly in space. A 2 x 2 mask is used because the goal is only to reduce the noise of the edge image and not remove relevant edge information. Equation (2-2) shows the 2 x 2 mask used for the average spatial filter.

$$\text{avg} = \frac{1}{4}\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \tag{2-2}$$

2.2.2 Image Binarization

The goal of image binarization is to remove the noise with low grayscale values. Since the average spatial filter diminishes the pixel intensity of the regions that have few pixels with high values, it is possible to eliminate these regions by means of applying a threshold value to the image. Image binarization for a given *Image* and *Threshold* values is given by the equation (2-3).

$$OutImage_{i,j} = \begin{cases} 0 & Image_{i,j} \le Threshold \\ 1 & Image_{i,j} > Threshold \end{cases} \tag{2-3}$$

2.2.3 Edge Thinning

Edge thinning is required when the border images are used in GHT since the computational cost can be drastically reduced by decreasing the number of points that are analyzed. To preserve the relevant information of the edge image, edge thinning should not remove endpoints nor cause excessive erosion of a region [15].

The edge thinning process in this project is given by defining the following sets based on the actual pixel content

$$\begin{aligned} same(i,j) &= \left\{ Grad_x(i,j), Grad_y(i,j) \middle| Grad_x(i,j) = Grad_y(i,j) \right\} \\ sum(i,j) &= \left\{ Grad_x(i,j), Grad_y(i,j) \middle| Grad_x(i,j) + Grad_y(i,j) = 255 \right\} \\ A(i,j) &= \left\{ Grad_x(i,j) \middle| 35 < Grad_x(i,j) \le 125 \right\} \\ B(i,j) &= \left\{ Grad_y(i,j) \middle| 35 < Grad_y(i,j) \le 125 \right\} \\ C(i,j) &= \left\{ Grad_x(i,j) \middle| 128 \le Grad_x(i,j) \le 220 \right\} \\ D(i,j) &= \left\{ Grad_y(i,j) \middle| 128 \le Grad_y(i,j) \le 220 \right\} \end{aligned} \tag{2-4}$$

Then, its corresponding indexes are defined as

$$\begin{aligned}
(i,j)_a &= \{(i,j) \in A(i,j) \cap B(i,j)\} \\
(i,j)_b &= \{(i,j) \in A(i,j) \cap D(i,j)\} \\
(i,j)_c &= \{(i,j) \in C(i,j) \cap B(i,j)\} \\
(i,j)_d &= \{(i,j) \in C(i,j) \cap D(i,j)\} \\
(i,j)_e &= \{(i,j) \in same(i,j) \cap (GradY(i,j) > 69)\} \\
(i,j)_f &= \{(i,j) \in sum(i,j) \cap (GradY(i,j) < 186)\}
\end{aligned} \tag{2-5}$$

Then, the set of indexes to be zeroed-out are defined as

$$(i,j)_{zero} = \{(i,j)_a \cup (i,j)_b \cup (i,j)_c \cup (i,j)_d \cup (i,j)_e \cup (i,j)_f\} \tag{2-6}$$

Finally, single-edge border images are given by

$$\begin{aligned}
Grad_y(i,j) &= \{0, Grad_y(i,j) \mid Grad_y(i,j) \in (i,j)_{zero}\} \\
Grad_x(i,j) &= \{0, Grad_x(i,j) \mid Grad_x(i,j) \in (i,j)_{zero}\}
\end{aligned} \tag{2-7}$$

For each pair i and j with $1 \le i \le rows$ and $1 \le j \le cols$.

2.3. Extracting Gradient Information

Extracting the gradient information of the template and the target images is performed before computing the GHT. The directional information θ_i is calculated using (2-8).

$$\theta(x,y) = \arctan\left(\frac{Grad_y}{Grad_x}\right) \tag{2-8}$$

$Grad_x$ and $Grad_y$ represent the first derivative of the image. These are calculated using equation (2-9).

$$\begin{bmatrix} Grad_x \\ Grad_y \end{bmatrix} = \begin{bmatrix} \frac{\partial A(x, y)}{\partial x} \\ \frac{\partial A(x, y)}{\partial y} \end{bmatrix} \tag{2-9}$$

Sobel and Prewitt gradient operators are the most commonly used gradient edge detectors [8]. Given that these operators are only an approximation of the continuous gradient, Ando's gradient operator is used [16]. By providing a higher peak in the GHT accumulator, this operator is known to perform better than Sobel and Prewitt [7]. The 3 x 3 optimum gradient operators are shown in equation (2-10).

$$\nabla_x = \frac{\partial}{\partial x} \approx \begin{bmatrix} -0.112737 & -0.274526 & -0.112737 \\ 0 & 0 & 0 \\ 0.112737 & 0.274526 & 0.112737 \end{bmatrix}$$
$$\nabla_y = \frac{\partial}{\partial y} \approx \begin{bmatrix} -0.112737 & 0 & 0.112737 \\ -0.274526 & 0 & 0.274526 \\ -0.112737 & 0 & 0.112737 \end{bmatrix} \tag{2-10}$$

2.4. Atmel® SAMV71Q21

In this section, an overview of the hardware selected will be presented, along with the most relevant features that are directly related to signal processing, which were used to achieve the maximum performance of the image processing library implemented in this project.

The Atmel® SAMV71Q21 is a high-performance flash microcontroller (MCU), a member of the SMART SAM V71 family of devices based on the 32-bit ARM® Cortex®-M7 RISC (5.04 CoreMark/MHz) processor with a floating point unit (FPU). Although the Cortex®-M processor

family is more focused on the lower end of the performance scale, these processors are still powerful when compared to other typical processors.

The device operates at a maximum speed of 300 MHz, it features 2048 kB of Flash, 16 kB of dual cache memory and 384 kB of SRAM. Fig. 2-3 shows the implementation of this MCU [17].

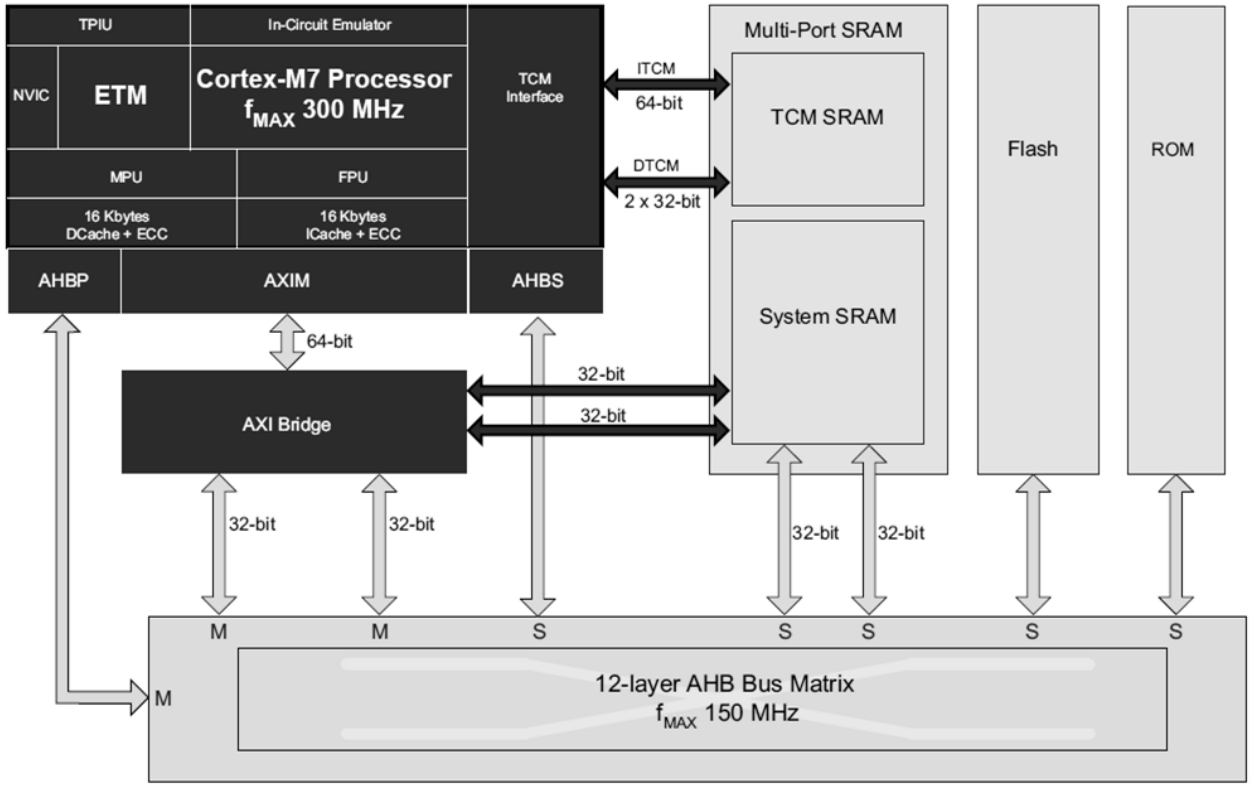


Fig. 2-3 Atmel® SAMV71Q21 - ARM® Cortex®-M7 processor implementation [18].

2.4.1 Maximizing Performance

The Atmel® SAMV71Q21 MCU implements the Cortex®-M7 processor according to the high-performance configuration recommended by ARM®. Table II shows the high-performance features available in the Atmel® SAMV71Q21 MCU.

Table II
Atmel® SAMV71Q21 performance features [17]

Feature	Atmel® SAMV71Q21
FPU	Single and Double Precision FPU
ITCM max size	128 kB
DTCM max size	128 kB
I-cache size	16 kB
D-cache size	16 kB
Multi-port SRAM	384 kB
AHB Peripheral (AHBP) size	512 MB
ECC support on caches	Implemented
MPU	16 regions
Interrupts	72
Debug watchpoints and breakpoints	4 data watchpoints + 8 breakpoints
ITM and DWT Trace	Implemented
ETM	ETM Instruction Trace only

2.4.2 Six-stage Superscalar Pipeline

The Cortex®-M7 has a six-stage superscalar pipeline with branch prediction that enables the processor to execute two instructions in parallel. The pipeline features an optional float pipeline so that floating point instructions can be dual-issued with integer instructions as well. Memory accesses are interleaved with computation to reduce latency. The processor features an integer MAC instruction execution that takes one clock cycle [17].

2.4.3 Multi-port SRAM

Multi-port SRAM in the Atmel® SAMV71Q21 MCU is 384 kB. This SRAM space operates at bus clock (i.e. processor clock / 2 = up to 150 MHz) and has four ports to optimize the bandwidth and latency. The purpose of the multi-port capability is to decrease the latency when several masters try to access the SRAM simultaneously. The integrated controller manages interleaved addressing of SRAM blocks so that another master will be able to have access on the next cycle [17].

The Tightly Coupled Memory (TCM) region is shared with the system's RAM region. The size of the TCM region can be configured through NVM bits, and the remaining SRAM size is automatically assigned to the RAM region. Fig. 2-3 shows the implementation of the SRAM.

2.4.4 Instruction and Data Cache

The Atmel® SAMV71Q21 MCU embeds an instruction and data cache (I-cache and D-cache) to compensate wait state penalty when executing code out from the external memory (typically flash). The size of I-cache and D-cache is 16 kB each [17].

2.4.5 Tightly Coupled Memories

Real-time applications require algorithms that run fast and deterministically in time. These applications must be protected against cache misses, interrupts, context swaps, and other run-time events. The Cortex®-M7 architecture provides a way to avoid the slow memory access time caused by the cache miss delay, bypassing the standard execution mechanism using TCM [19].

The TCM controller provides a direct channel between the processor and two memory areas: Instruction TCM (ITCM) and Data TCM (DTCM). The size of these memories is 128 kB each for the Atmel® SAMV71Q21 MCU. Instructions and data located in the TCM can be directly accessed at processor speed (i.e.: up to 300 MHz) with no wait state penalty, as opposed to the other memories such as the flash, which is accessed at bus speed through the AXI master interface [18].

The purpose of the TCM memories is to store the critical part of the code, which needs to be processed as fast as possible. TCM memory contents are not cached. TCM memory is directly connected to the Cortex®-M7 core by a bus. It can be accessed at similar speeds as accessing cache without the penalty of a cache miss and cache coherence issues. Table III highlights the key differences between cache and TCM.

Table III
TCM vs. cache memory [18]

Tightly Coupled Memory	Cache Memory
TCM is a memory accessed by a dedicated connection from the core. There are two dedicated connections from Cortex®-M7 to the internal SRAM – for Instruction TCM and another for Data TCM.	Cache memory is RAM memory integrated inside the Cortex®-M7 core itself.
TCM is part of the system memory map with a definite start address. The size of the TCM determines the end address.	Cache memory is not part of the system memory map. It does not have a physical memory address.
A programmer can decide the content to be stored in TCM at compilation time.	A control logic determines what is stored in cache memory.
The TCM memory is directly accessible to software.	During program execution, the cache is stored with instructions or data fetched from memory to the CPU.
TCM can be accessed both by CPU and by DMA.	Cache memory serves as an intermediate buffer between the processor and memory to reduce memory access time. The number of cycles needed to access a memory location differs for a cache-hit and a cache-miss.

The code that will be executed out of the TCM must be identified by the programmer. When preparing a software build, the programmer will need to identify which code segments and data blocks should be allocated to the TCM by modifying the linker script. For this project, the image processing library routines will be allocated in ITCM.

2.4.6 Internal Flash Memory

Internal flash memory is accessed through the AXI Master and AHB matrix at bus clock. Wait states must be added depending on the operating frequency (up to five wait states when the processor is running at a maximum frequency), I-cache and D-cache can be used to compensate for the high memory access time [17].

2.4.7 Floating Point Unit

The Atmel® SAMV71Q21 MCU integrates a double-precision FPU that supports the ARMv7 VFPv5 architecture. It is tightly integrated to the ARM® Cortex®-M7 processor pipeline. It provides trapless execution and is optimized for scalar operations [17].

3. Methodology

In this section, the most important elements for the implementation of the image processing library are presented. The system configuration applied in this project to achieve the maximum performance and the software routines that comprise this library are described in detail.

3.1. Implementation of the Image Processing Library

In order to achieve the maximum performance of the image processing library, the GNU ARM assembly language was initially considered in order to take full advantage of the low-level signal processing functions offered by the ARM® Cortex®-M7 architecture. Given that the equivalent implementation of a 2 x 2 spatial filter was faster when implemented in GNU ARM C than in assembly language, the GNU ARM C toolchain was used. To obtain the maximum performance and improve the object size, the compiler optimization was set to O3.

The performance of the library was determined by two metrics, execution time and object size. Execution time was measured using the clock cycle counter of the Data Watchpoint and Trace (DWT) module in the MCU. Clock count was measured only for the core operations of the routines, neglecting setup and teardown sections. Object size was measured by obtaining the address offsets of the image processing routines in the memory map.

The target image was precompiled for validating the implementation of the routines, and it was located in the read-only memory section (internal flash memory). In a real application, this buffer would be obtained from a camera through an image sensor interface. The R-table buffers for the template image were also located in the internal flash, provided these values will be constant.

The image processing buffers are located in RAM since low memory access time for the image buffer is critical for achieving a maximum performance. The image processing routines

were validated by dumping the memory of these buffers and converting the binary data using Matlab to either an image or a mesh graph for the case of the output of the GHT routine.

3.2. Atmel® SAMV71Q21 System Configuration

The Atmel® SAMV71Q21 MCU was configured as follows to achieve maximum performance.

- Operating frequency: 300 MHz.
- TCM enabled; ITCM size: 128 kB; DTCM size: 128 kB.
- Hardware FPU enabled.
- Instruction and data cache enabled.
- Software to run from internal flash memory.

3.3. Image Processing Library

The routines that comprise the Atmel® SMART SAM V71 image processing library will be reviewed in this section.

- Integer Spatial Filtering 2 x 2 (Correlation)
- Integer Spatial Filtering 3 x 3 (Correlation)
- Image Binarization
- Image Subtraction
- Image Gradient Orientation
- Edge Thinning
- Generalized Hough Transform

A brief description of the routines along with application examples and the proposed implementation algorithm will be presented. The implementation of these routines is based on a previous work for the Freescale® MPC5561 microcontroller [14].

3.3.1 Integer Spatial Filtering 2 x 2

Description: Filtering of an 8-bit unsigned integer image based on spatial correlation with floating-point masks of size 2 x 2. The result is stored into an array of 8-bit unsigned integers representing the output image.

Application: Using an average mask, this routine is used to soften the lines of an image. Fig. 3-1 shows an example application.

Algorithm:

- Set *Image* to be a *rows* x *cols* array.
- Set *Mask* to be a 2 x 2 filtering mask.
- *OutImage* is the spatial correlation (filtering) of *Image* using *Mask* given by

(a) (b)

Fig. 3-1 Spatial filtering of a sample image using a 2 x 2 average mask. (a) Original; (b) Filtered image.

$$OutImage_{i,j+1} = Image_{i,j} \cdot Mask_{1,1} + Image_{i,j+1} \cdot Mask_{1,2} + Image_{i+1,j} \cdot Mask_{2,1} + Image_{i+1,j+1} \cdot Mask_{2,2} \quad (3\text{-}1)$$

For each pair *i* and *j* with $1 \leq i \leq rows - 1$ and $1 \leq j \leq cols - 1$.

Note that first column and last row of the output images are not calculated. These values are left unchanged.

3.3.2 Integer Spatial Filtering 3 x 3

Description: Filtering of an 8-bit unsigned integer image based on spatial correlation with floating-point masks of size 3 x 3. The result is stored into an array of 8-bit unsigned integers representing the output image.

Application: Examples of commonly used filtering masks are the following:

- Gaussian
- Ando
- Average
- Prewitt

Fig. 3-2 shows an example application using a Gaussian mask.

Algorithm:

- Set *Image* to be a *rows* x *cols* array.
- Set *Mask* to be a 3 x 3 filtering mask.
- *OutImage* is the spatial correlation (filtering) of *Image* using *Mask* given by

$$OutImage_{i+1,j+1} = Image_{i,j} \cdot Mask_{1,1} + \ldots + Image_{i,j+2} \cdot Mask_{1,3} + Image_{i+1,j} \cdot Mask_{2,1} + \ldots$$
$$+ Image_{i+1,j+2} \cdot Mask_{2,3} + Image_{i+2,j} \cdot Mask_{3,1} + \ldots \quad (3\text{-}2)$$
$$+ Image_{i+2,j+2} \cdot Mask_{3,3}$$

For each pair i and j with $1 \leq i \leq rows - 2$ and $1 \leq j \leq cols - 2$.

Note that first and last columns, as well as first and last rows of the output image, are not calculated. These values are left unchanged.

(a) (b)

Fig. 3-2 Spatial filtering of a sample image using a 3 x 3 Gaussian mask. (a) Original; (b) Filtered image.

3.3.3 Image Binarization

Description: This function computes the binary representation of an 8-bit unsigned integer image using a thresholding approach. The binary representation is stored into an array of 8-bit unsigned integers representing the output image.

Application: By modifying the threshold, this function can erase undesired information (e.g. noise) of an image. Fig. 3-3 shows an example application.

Algorithm:

- Set *Image* to be a *rows* x *cols* array.
- Set *Threshold* to be a reference value.
- *OutImage* is the binary representation of *Image* given by

$$OutImage_{i,j} = \begin{cases} 0 & Image_{i,j} \le Threshold \\ 255 & Image_{i,j} > Threshold \end{cases} \quad (3\text{-}3)$$

For each pair *i* and *j* with $1 \le i \le rows$ and $1 \le j \le cols$.

(a) (b)

Fig. 3-3 Binarization of a sample image using a threshold value of 128. (a) Original; (b) Binary image.

3.3.4 Image Subtraction

Description: Pixel by pixel subtraction of 8-bit unsigned integer images. The result is stored into an array of 8-bit unsigned integers representing the output image.

Application: This function can highlight features that differ from an image to another. For example, a Gaussian filtered version of an image can be subtracted from the original image to perform an unsharp masking operation. Fig. 3-4 shows an example application.

Algorithm: If *ImageA* and *ImageB* are arrays of identical dimensions *rows* x *cols*, then their subtraction is a *rows* x *cols* array denoted by *OutImage = ImageA - ImageB*. The subtraction is given by

$$OutImage_{i,j} = ImageA_{i,j} - ImageB_{i,j} \tag{3-4}$$

For each pair *i* and *j* with $1 \leq i \leq rows$ and $1 \leq j \leq cols$.

3.3.5 Image Gradient Orientation

Description: Compute directional information (θ) based on a discrete calculation of the arc tangent of the gradient operators of the image. Resulting gradient orientation image is stored into an array of 8-bit unsigned integers.

(a) (b) (c)

Fig. 3-4 Image subtraction application (a) – (b) = (c). (a) Original; (b) Gaussian filtered image; (c) Subtracted image (edge image).

Algorithm: The directional information of an image is given by obtained by applying a spatial filter of a 3 x 3 mask, using the Ando's gradient operator shown in equation (2-10). Table IV shows discrete values of tan(θ) in increments of 10°.

3.3.6 Edge Thinning

Description: This function merges dual-edge borders generated using directional gradient masks (such as Ando's directional masks) into single-edge borders. The resulting single-edge border images are stored into same 8-bit unsigned integer input arrays.

Application: After using Ando's directional gradient masks to obtain $Grad_x$ and $Grad_y$, 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 be processed.

Algorithm: Equations (2-4), (2-5), (2-6) and (2-7) show the edge thinning algorithm implemented for this project.

3.3.7 Generalized Hough Transform

Table IV
Discrete values for tan(θ)

Degrees	Radians	tan(θ)	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

Description: Alpha (α) information must be stored in the 16-bit arrays *AlphaSine* and *AlphaCosine* in the form of sin(α) and cos(α) multiplied by a factor of 256, respectively. Theta (θ) information is stored in *ThetaLength* and *ThetaIndex* and must contain the number of elements with the same orientation and their absolute index within the template images, respectively.

Application: The user can take advantage of the Hough transform function to implement template segmentation techniques. One application is recognition of traffic signs along the road by an onboard vehicle computer paired with an imaging sensor.

Algorithm: Fig. 3-5 shows the algorithm for the typical implementation of the GHT. The template image's R-table is used to compute candidate reference points in the target image. The accumulator is updated in every iteration.

3.3.8 Image Maximum Search

Description: Search for the maximum value of an 8-bit unsigned integer image of size *rows* x *cols*. The maximum value and its coordinates in the form of 32-bit unsigned integers are the output values.

Application: 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. One 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.

Algorithm:

$$MaxValue(i, j) = \max\left\{Image_{i,j}\right\} \tag{3-5}$$

For each pair *i* and *j* with $1 \leq i \leq rows$ and $1 \leq j \leq cols$.

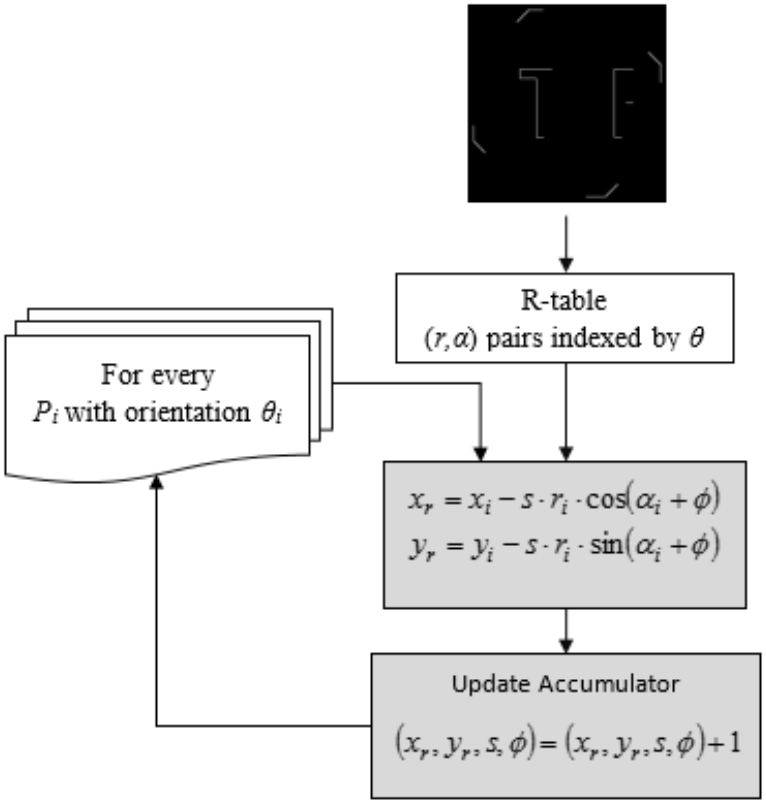


Fig. 3-5 Typical implementation of the Generalized Hough Transform algorithm [14].

3.4. Image Processing Library performance on Freescale® MPC5561

Table V shows the performance measurements for the image processing library routines implemented in this project for a different hardware, a Freescale® MPC5561 MCU.

Although the Freescale® MPC5561 is based on a different architecture, this microcontroller is designed for ADAS applications, specifically for those that use radar or image sensors. Its characteristics make it a good choice for digital signal processing applications, among the most relevant are [20]:

- Core

    - 132 MHz PowerPC ISA e200 Core
        - SIMD module for DSP and floating-point features
    - 32 kB Unified Cache
- Memory
    - 1 MB RWW Flash with ECC
    - 192 kB SRAM with ECC
- System
    - DMA Controller
    - Interrupt Controller
    - External Bus Interface

Note that the execution time of the image processing routines marked with an asterisk (*) vary depending upon image content. These values represent the average performance obtained during development and testing of a limited number of images.

Table V
Freescale® MPC5561 performance of image processing routines [14]

Function name	Object size (Bytes)	Execution time (Clock cycle / Pixel)
Integer Spatial Filtering 2 x 2	316	13.8
Integer Spatial Filtering 3 x 3	732	20.7
Image Binarization	300	5.8
Image Subtraction	332	3.8
Image Gradient Orientation*	704	8.7
Edge Thinning*	448	8.9
Generalized Hough Transform*	540	51.0
Image Maximum Search	268	9.8

4. Results

The image processing library described in the previous section was used to build a TSR application. During this process, the performance of the image processing routines was documented with the objective of benchmarking with the implementation for the Freescale® MPC5561.

4.1. Traffic Sign Recognition Application

The block diagram depicting the high-level approach used for the TSR application is shown in Fig. 4-1. Each gray block represents a routine of the image processing library implemented for this project. The image acquisition can be performed using a camera module connected to the Atmel® SAM V71 Xplained Ultra evaluation kit's image sensor interface.

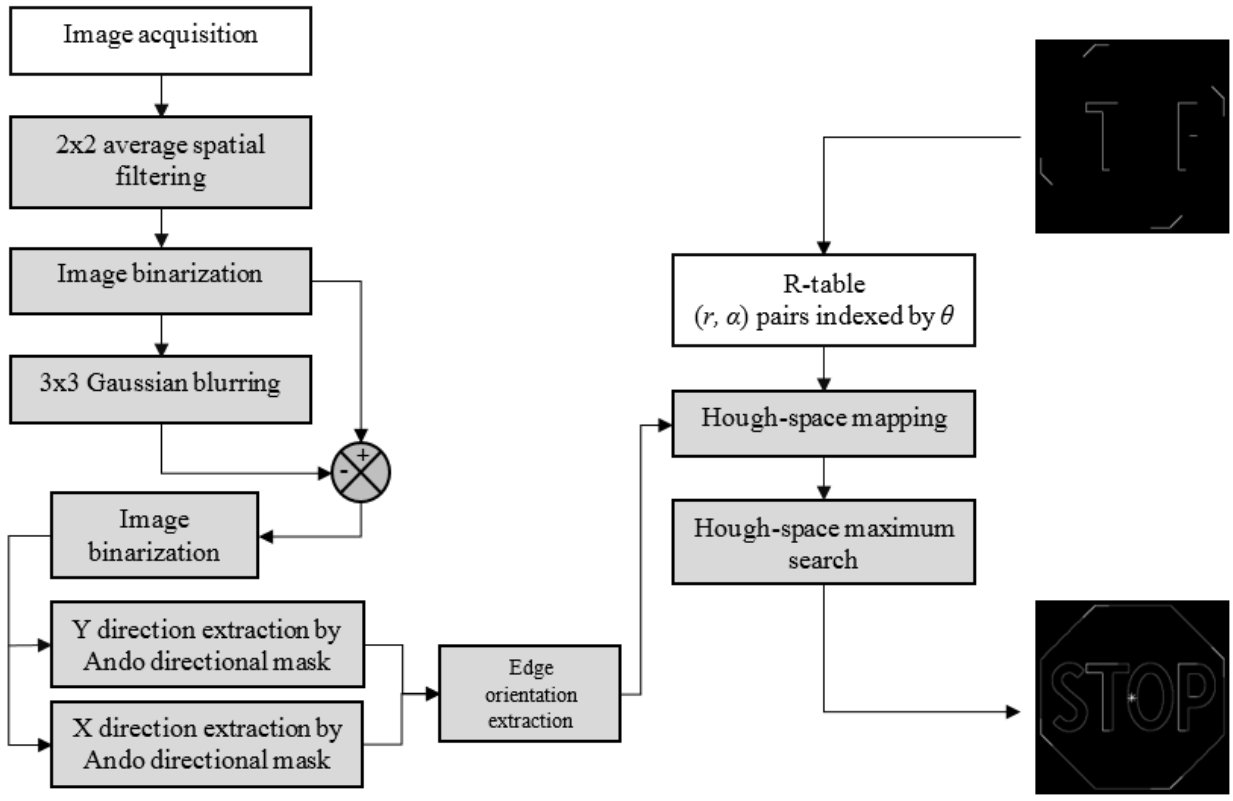
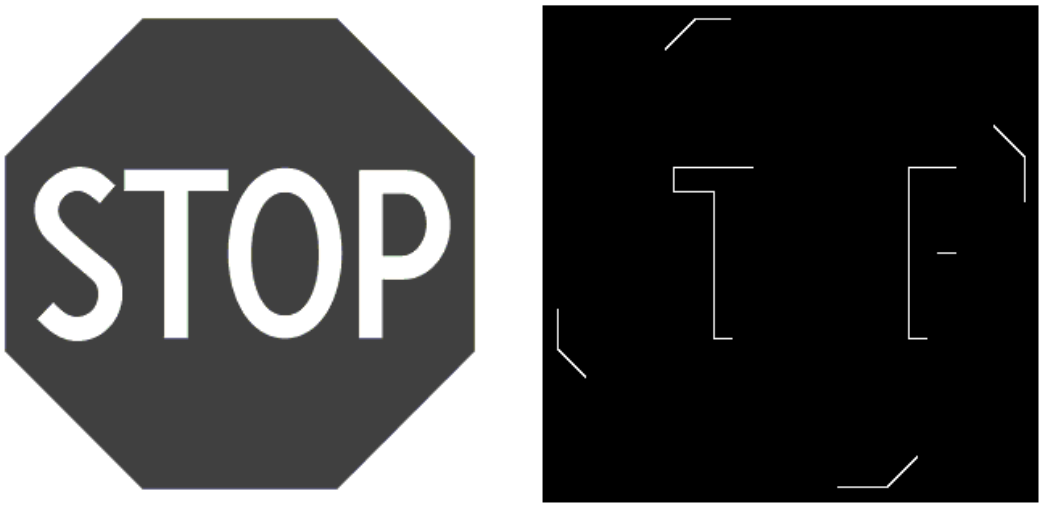


Fig. 4-1 Block diagram of the traffic sign recognition application [21].

The R-table for the template image was generated in advance using a Matlab script. The output of the algorithm is the coordinates of the point with the highest number of votes of the template image in the target image.

The target and template images selected as an example for this application, are shown in Fig. 4-2. The target image is a stop sign; this image was preprocessed to convert it to grayscale in order to be used with the image processing library. The template image was created by extracting the most relevant border information of the stop sign. It is important to note that the greater the number of pixels in this template image, the greater the computational cost of the GHT.



(a) (b)

Fig. 4-2 Traffic Sign Recognition application input. (a) Target image; (b) Template image.

The image processing starts after acquiring the image, in the first place, a 2 x 2 average spatial filter and binarization is applied. The output of these two processes is an image with low noise. Then, in another memory buffer, a blurred version of the binarized image is stored. This image is created using a 3 x 3 Gaussian spatial filter. With these two images, an unsharp masking technique is applied, subtracting the blurred binarized image to the binarized image. The resulting image is binarized again to remove the noise introduced in the previous stages. Fig. 4-3 shows the output images in each of the phases described in this paragraph.

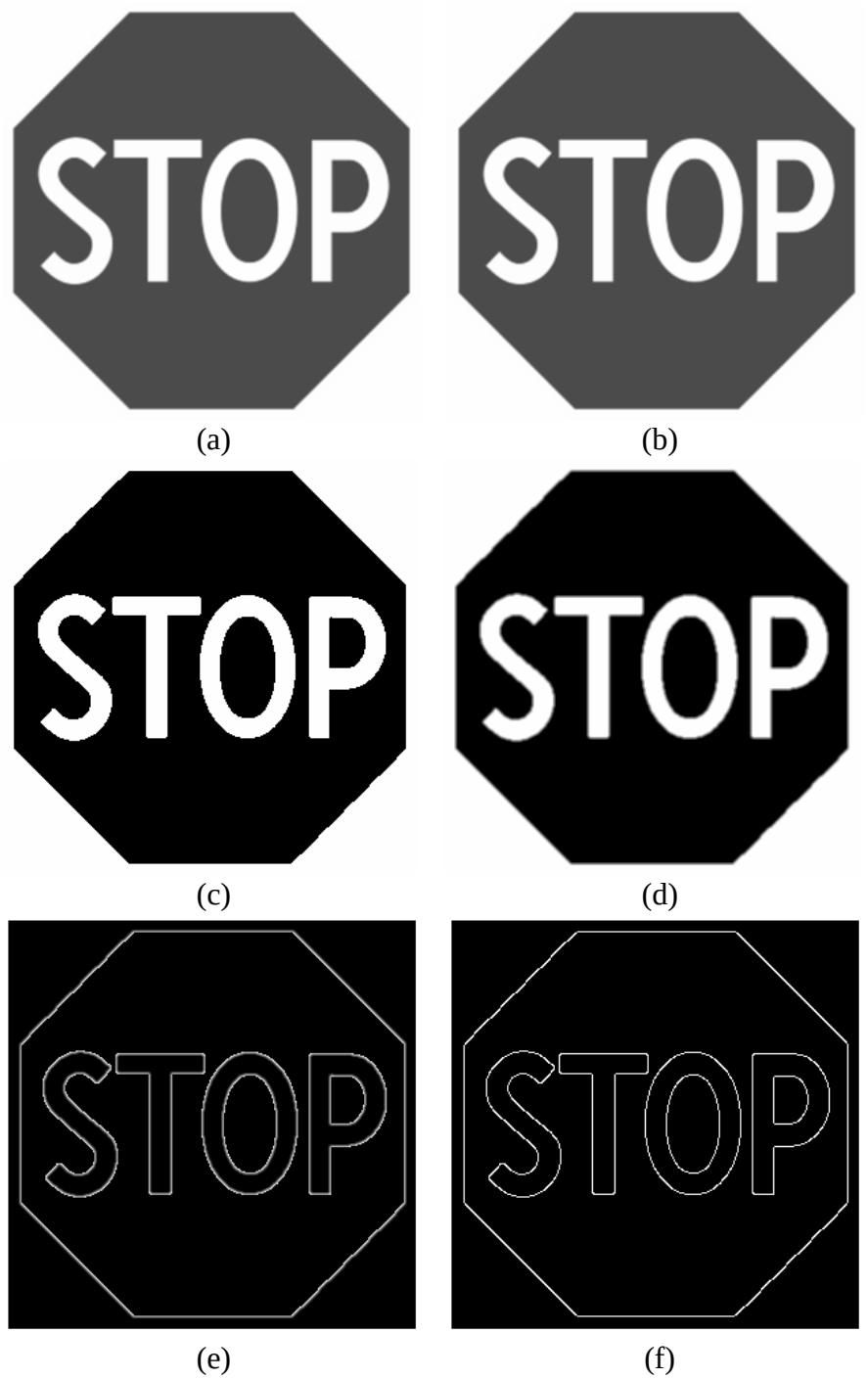
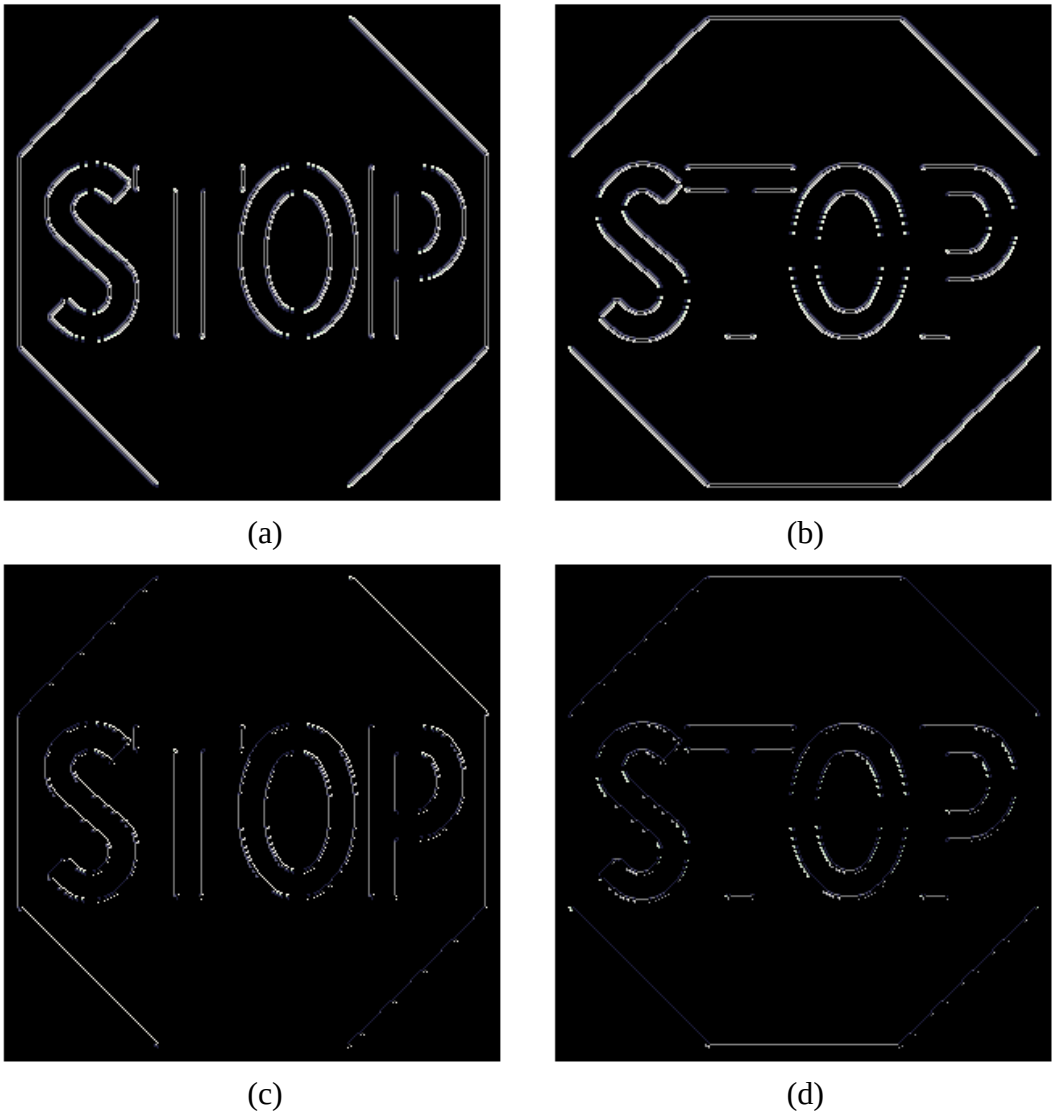


Fig. 4-3 Traffic Sign Recognition processing part I. (a) Original image; (b) 2 x 2 average spatial filtering; (c) Image binarization; (d) 3 x 3 Gaussian spatial filtering; (e) Unsharp masking image; (f) Unsharp masking image after image binarization.

At this point, the working buffer contains an edge image with little or no noise, and the next step is to apply Ando's directional mask to obtain the *x* and *y* directional information of the

image. Then, the edge thinning routine is invoked for the output images with the purpose of reducing the number of edges in the image. Note that the amount of border information in the target image also impacts the computational cost. Fig. 4-4 shows the output images for these routines.



(a) (b)

(c) (d)

Fig. 4-4 Traffic Sign Recognition processing part II. (a) Image gradient *y*; (b) Image gradient *x*; (c) Image gradient *y* after edge thinning routine; (d) Image gradient *x* after edge thinning routine.

A new buffer containing the discrete directional information of the image given by the equation (2-8) and Table IV is fed to the GHT routine along with the R-table of the template image

in the form of three arrays containing the pre-calculated r, $\sin(\alpha)$, and $\cos(\alpha)$ for a given orientation. These arrays are obtained in advance using a Matlab script that receives an image as the input argument and creates a C header file that contains these image template definitions.

The output of the GHT function is a buffer containing the Hough accumulator. The maximum value in this buffer is located using the image maximum search routine. Fig. 4-5 shows the composite image of the target and template images, showing the reference point of the template obtained from the GHT routine.

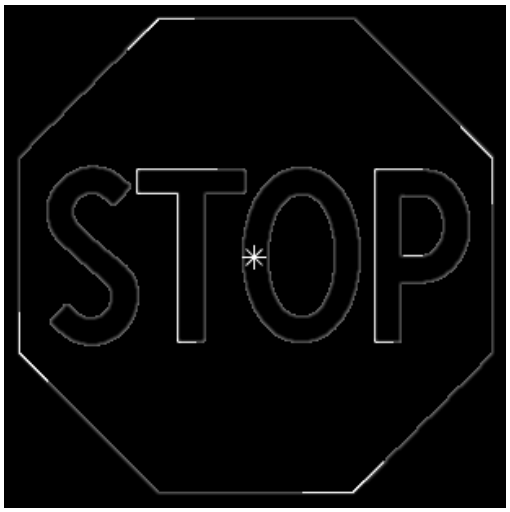


Fig. 4-5 Location of the template image's reference point the target image.

Fig. 4-6 shows the mesh graph of the Hough accumulator. It is worth noting that the vote count that is shown in the Hough accumulator in Fig. 4-6 represents a perfect match of the template image in the target image. By modifying the scale factor of the template image, no peak at the Hough accumulator was observed. An example is shown in Fig. 4-7.

In a TSR application, the difference in the Hough accumulator peak value is used as a software flag to indicate a template match in the target image.

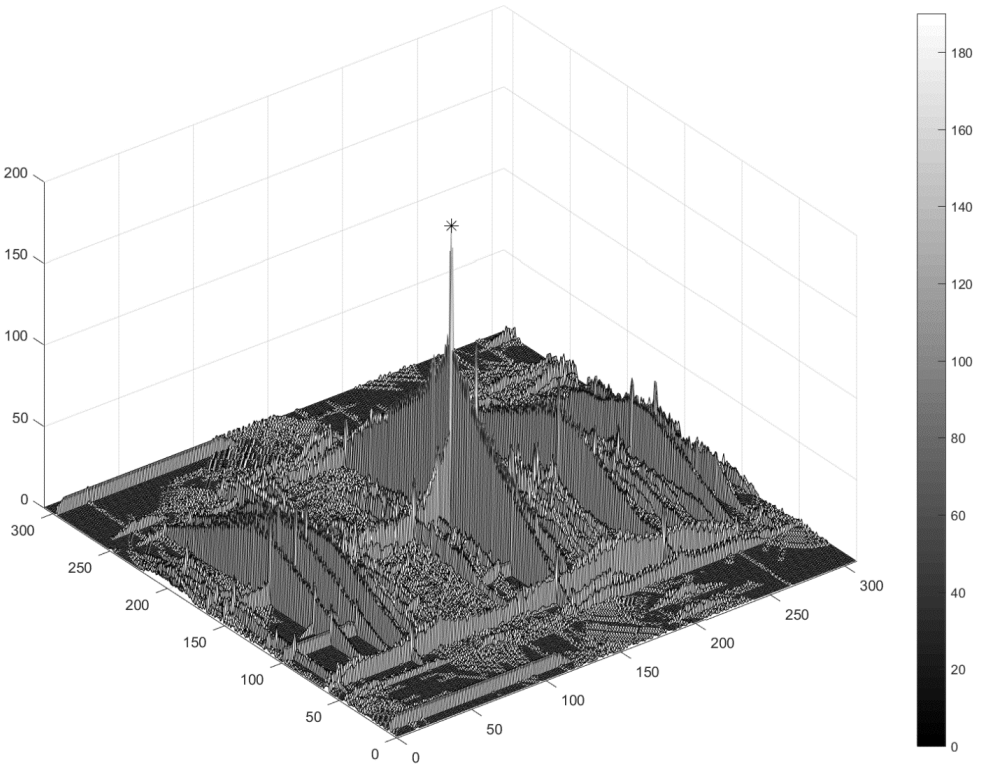


Fig. 4-6 Perfect match Hough accumulator.

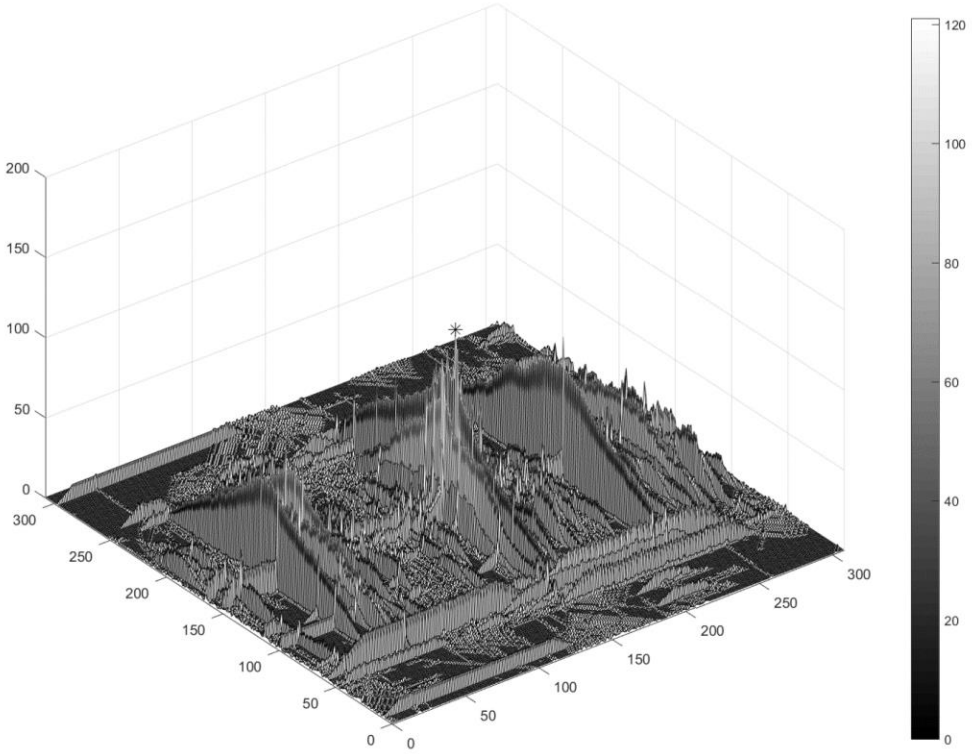


Fig. 4-7 Hough accumulator when the image template shown in Fig. 4-2 has been scaled.

4.2. Image Processing Library performance on Atmel® SAMV71Q21

The performance of the image processing library implemented for the Atmel® SAMV71Q21 is shown in Table VI. The values in this table can be compared to those in Table V, except for the routines that are marked with an asterisk (*), since the execution time for those routines is directly related to the image content. Nevertheless, the content and size of the template and target images were similar to those used for the Freescale® MPC5561 image processing library implementation so that the execution time could be benchmarked.

Table VI
Atmel® SAMV71Q21 performance of image processing routines

Function name	Object size (Bytes)	Execution time (Clock cycle / Pixel)
Integer Spatial Filtering 2 x 2	256	9.8

Integer Spatial Filtering 3 x 3	424	19.6
Image Binarization	96	5.7
Image Subtraction	308	3.0
Image Gradient Orientation*	224	5.4
Edge Thinning*	504	4.6
Generalized Hough Transform*	616	66.5
Image Maximum Search	224	6.5

The TSR application shown in Fig. 4-1 implemented with the Atmel® SAMV71Q21 MCU, can process up to 19 frames of 308 x 308 pixels per second. The signal processing rate varies depending on the image processing application implementation.

4.3. Image Processing Library Benchmarking

The execution time and object size of the image processing library routines implemented for the Atmel® SAMV71Q21 were compared with the Freescale® MPC5561 implementation. The result is shown in Fig. 4-8.

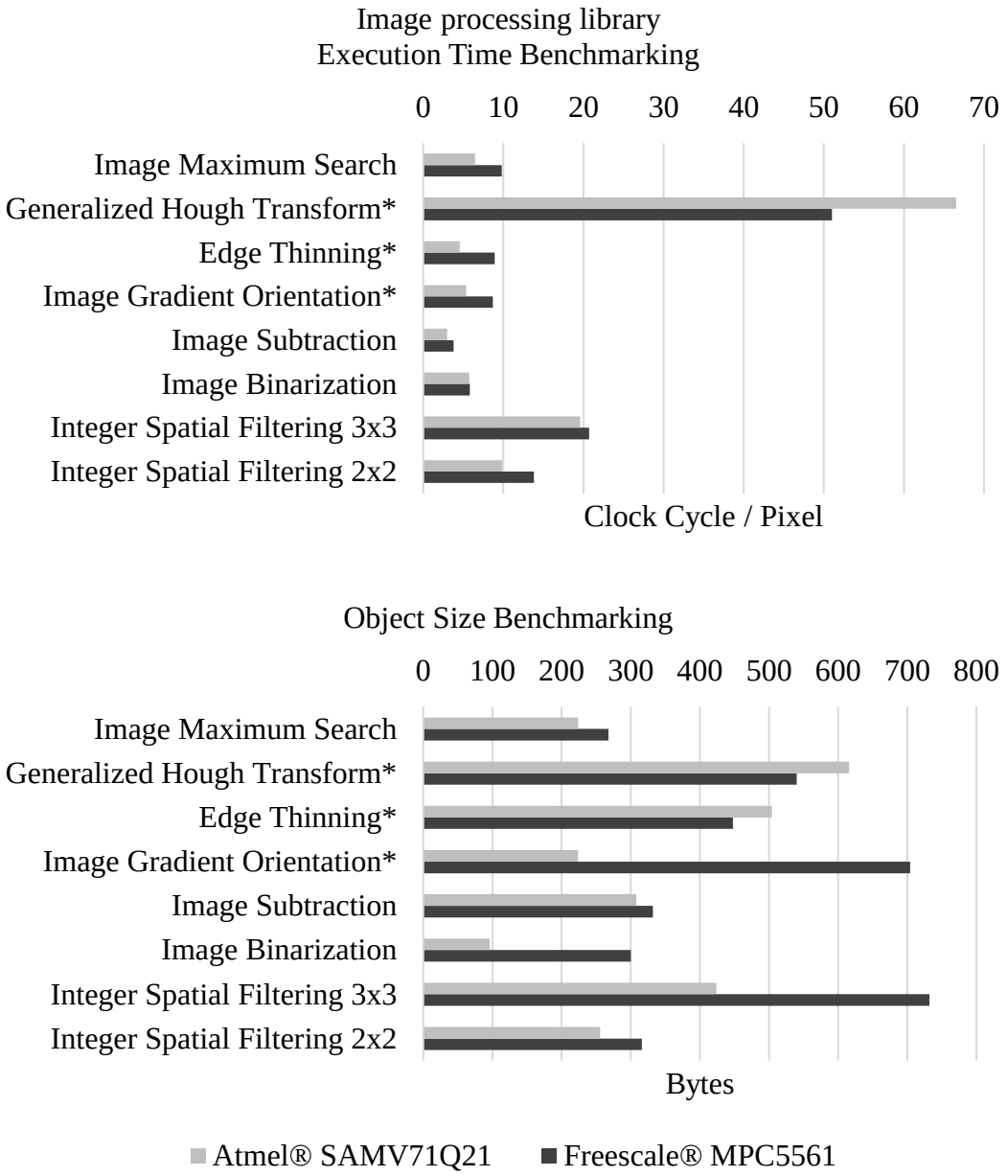


Fig. 4-8 Performance benchmarking for the image processing library implementation for the Atmel® SAMV71Q21 and the Freescale® MPC5561 microcontrollers. The execution time of the image processing routines marked with an asterisk (*) vary depending upon the image content.

By using the GNU ARM C compiler optimization, the object size is in average 25% smaller. This enables the embedded system for more robust image processing applications. Yet, there are two cases where the resulting objects are 13% larger compared to the reference implementation. These cases correspond to the Edge Thinning and GHT routines.

Execution time is measured in MCU clock cycles per pixel, in average the implementation for Atmel® SAMV71Q21 is 18% faster than the reference. This is achieved by the GNU ARM C compiler optimization techniques such as inline functions, loop unrolling, and automatic vectorization.

5. Discussion

The performance of the image processing library implemented in this project has been validated with a TSR application. The results show that the execution time and binary object size were improved compared to the reference implementation.

Benchmarking results were positive for object size provided that the resulting binary objects are in average 25% smaller than the Freescale® MPC5561 implementation. This difference is critical for the suitability of the image processing library in embedded applications. The negative difference of the two image processing routines that resulted in objects 13% larger (GHT and edge thinning) is a reasonable tradeoff, given that by implementing these routines in a high-level programming language such as C, the maintainability of the library increases compared to the implementation in assembly language. Furthermore, the average gain in space is 37% in the rest of the routines.

Execution time benchmarking results were also positive with an 18% average gain in performance. It is important to note the role of the GNU ARM C compiler optimizations in this result. These optimizations offer the possibility of a better performance even though the Atmel® SAMV71Q21 is not a signal processing microcontroller such as Freescale® MPC5561. This gain is only achievable when the compiler optimizations are enabled.

The performance of the TSR application example presented in this project is superior for the Atmel® SAMV71Q21 since the execution time is minor and the processor's clock speed is 2.3 times faster. Depending on the size of the frames, the optimized image processing library is suitable for real-time applications by obtaining an fps rate up to 24. It is worth mentioning that the presented TSR implementation can still be improved and thus, the fps rate can increase.

Other image processing applications can also benefit from the high performance achieved through this implementation. For instance, according to Table VI, the spatial filtering routine with

a 3 x 3 mask can process 30 frames of 640 x 480 pixels in 602 ms. This represents a gain of 39% in execution time compared with the FPGA implementation presented by Vega [5]. Nevertheless, other applications still provide a better performance mostly by reducing the indexing time; for instance, the 3 x 3 spatial filter presented by Park is 60% faster [6]. However, the reduced cost when using the main MCU instead of a dedicated FPGA module as an image processing core must be considered.

Regarding the implementation details of the image processing routines, the PowerPC assembly implementation makes use of programming optimization techniques. For instance, using SIMD instructions to perform up to four arithmetic operations in a single clock cycle, employing dedicated branching registers to avoid unnecessary comparison instructions in every iteration, and selecting the right order for memory access and computation instructions to avoid pipeline stalls. Given that these techniques are not automatic, they require relevant programming expertise and multiple fine-tuning iterations until the maximum performance is achieved.

On the other hand, the ARM® Cortex®-M7 family cannot fully benefit from the SIMD instruction set due to the reduced number of general purpose registers and the lack of special signal processing registers. These registers are required to load the multiple operands of SIMD instructions, and depending on the algorithm, an overhead can be created for every SIMD computation. Also, this architecture does not include special branching registers and the GNU ARM C compiler optimization already does a great job of interleaving the memory access and computing instructions.

The C programming language with GNU ARM C compiler optimization was the toolchain selected to implement the image processing routines since the assembly language optimization techniques did not achieve the maximum performance in the Atmel® SAMV71Q21, but instead, increased the complexity of the library. Moreover, due to the portability of the C programming language, this library can be easily adapted to other systems even though achieving the maximum performance out of every system, might require a proper configuration.

Conclusions

An optimized image processing library was developed with all the necessary routines to perform image segmentation. This software library was implemented for the Atmel® SAMV71Q21 MCU and can be used in other members of the ARM® Cortex®-M7 family. The software library was verified using a TSR application by testing with multiple patterns of different sizes. The performance of these routines was measured on run time and it provided an average gain in execution time of 18% compared to the Freescale® MPC5561 implementation.

The performance results of the spatial filter routines are competitive with the current FPGA implementations. When using all the image processing routines in a TSR application the results were also improved compared to the reference implementation, doubling the fps rate. The maximum performance configuration of the Atmel® SAMV71Q21 MCU is critical for this achievement. By enabling the data cache, instruction cache, and TCM, the working image buffers can be accessed at the processor's clock speed. Besides, the image processing fps rate can still be increased by improving the TSR application implementation.

During the development phase, it was noted that the main drawback of GHT was its execution time. It is critical to select the appropriate template image so that the edge information is enough to find a match in the target image without having a high computational cost. It is also important to have an automated process for obtaining the template information.

The code for the image processing routines was written in the C programming language, leveraging GNU ARM C compiler optimizations to obtain maximum performance and a minimum object size. Given the ubiquity of this programming language, it is possible to use this library in other systems, not limited exclusively to the embedded realm; although it is possible that the performance will not be the best in every system without proper configuration. Using a high-level programming language also increases the maintainability and extensibility of this library compared to the implementation in assembly language.

References

[1] A. Tigadi, R. Gujanatti, G. Anil and K. Belagavi, "Advanced Driver Assistance Systems," *Int. J. of Eng. Research and General Sci.*, vol. Volume 4, no. Issue 3, 2016.

[2] K. A. Brookhuis, D. D. Waard and W. H. Janssen, "Behavioural Impacts of Advanced Driver Assistance Systems—An Overview," *European J. Transport and Infrastructure Research*, vol. 1, no. 3, pp. pp. 245 - 254, 2001.

[3] A. Lorsakul and S. Jackrit, "Traffic Sign Recognition Using Neural Network on OpenCV: Toward Intelligent Vehicle/Driver Assistance System," in *4th Int. Conf. on Ubiquitous Robots and Ambient Intelligence*, 2007.

[4] K. Dyczkowski, P. Gadecki and A. Kułakowski, "Traffic sign recognition system," in *World Conf. on Soft Computing*, San Francisco, USA, 2011.

[5] M. A. Vega-Rodríguez, J. M. Sánchez-Pérez and J. A. Gómez-Pulido, "An FPGA-based implementation for median filter meeting the real-time requirements of automated visual inspection systems," in *10th Mediterranean Conf. Control and Automation*, 2002.

[6] H. D. Park and J. Jae Wook, "FPGA design and implementation of edge enhancement by using 3×3 mask filter," in *Ind. Technology (ICIT), 2014 IEEE Int. Conf.*, 2014.

[7] A. Tezmol, H. Sari-sarraf, A. Mitra, R. Long and A. Gururajan, "Customized Hough transform for robust segmentation of the cervical vertebrae from x-ray images," in *Southwest Symposium on Image Analysis and Interpretation*, 2001.

[8] A. C. Bovik, The essential guide to video processing, Academic Press, 2009.

[9] G. A. Baxes, Digital image processing: principles and applications, New York: Wiley, 1994.

[10] D. H. Ballard, "Generalizing the Hough transform to detect arbitrary shapes," *Pattern recognition*, vol. 13, pp. 111-122, 1981.

[11] K. Gundolf, M. Vahl, J. Sarcher and M. Schaeferling, "A configurable architecture for the generalized hough transform applied to the analysis of huge aerial images and to traffic sign detection," in *ReConFigurable Computing and FPGAs (ReConFig), 2016 Int. Conf.*, 2016.

[12] S. R. Geninatti, J. I. Benavides Benítez, M. Hernández Calviño, N. Guil Mata and J. Gómez Luna, "FPGA implementation of the generalized Hough transform," in *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. Int. Conf.*, 2009.

[13] F. Schumacher and T. Greiner, "Two stage Real-Time stereo correspondence algorithm and FPGA architecture using a modified Generalized Hough transform," in *Systems, Signals and Image Processing (IWSSIP), 2014 Int. Conf.*, 2014.

[14] Freescale Semiconductor, *Appl. Note 3806,* 2009, pp. 3-21.

[15] R. C. Gonzalez and R. E. Woods, Image processing, Addison-Wesley, 1993.

[16] S. Ando, "Consistent Gradient Operators," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 22, no. 3, pp. 252-255, 2000.

[17] Atmel, *Appl. Note 44047,* 2016, pp. 3-15.

[18] Atmel, *Appl. Note 42510,* 2015, pp. 3-11.

[19] J. Wilbrink and L. Perdigon, "Run Blazingly Fast Algorithms with Cortex-M7 Tightly Coupled Memories," 2015.

[20] Freescale, *MPC5561 Microcontroller Data Sheet,* 2012.

[21] A. Tézmol and J. Shockey, "Enabling Automotive Vision Systems with the MPC5561 Microcontroller," in *Freescale Technology Forum*, 2008.

Appendices

A. IMAGE PROCESSING LIBRARY SOURCE CODE

```
/*****************************************************************************/
/**
  \file       ipl.c
  \brief      Image processing library
  \author     Roberto Ortega
              Based on the ASM implementation by Abraham Tezmol
  \version    1.0
  \date       2/January/2017
  */
/*****************************************************************************/

/*****************************************************************************
 * Include files
 *****************************************************************************/

/** board definitions */
#include "board.h"

/** standard types */
#include <stdint.h>

/*****************************************************************************
 * Definition of module wide VARIABLEs
 *****************************************************************************/

/* Public variable to store the clock cycle count of the last IPL API call
 * when IPL_MEASURE_CLK_CYCLES is defined */
uint32_t ipl_clk_cycles = 0;

/*****************************************************************************
 * Declaration of module wide FUNCTIONs
 *****************************************************************************/

/*****************************************************************************
 * Definition of module wide MACROs / #DEFINE-CONSTANTs
 *****************************************************************************/

/* Enable this switch to copy the pixels for first and last rows and columns
```

```
 *  for the spatial filter functions - this has an impact on performance.
 *  If disabled, the image_delete function removes the last two columns of the
 *  image to avoid unnecessary calculation in later stages of the process.
 */
#define IPL_COPY_PIXELS_FROM_ORIGINAL_IMG

/* Enable this switch to measure the execution time of each function call.
 *  This value is stored in the global function ipl_clk_cycles and is valid
 *  until the next call of the IPL API.
 */
#define IPL_MEASURE_CLK_CYCLES

#ifdef IPL_MEASURE_CLK_CYCLES
    #define IPL_MEASURE_RESET       RESET_CYCLE_COUNTER()
    #define IPL_MEASURE_GET         GET_CYCLE_COUNTER(ipl_clk_cycles)
#else
    #define IPL_MEASURE_RESET
    #define IPL_MEASURE_GET
#endif

/* --- Image Processing Library algorithm constants --------------------------*/

#define IPL_MASK_SCALE_FACTOR                       0x00010000  /* 65536 */
#define IPL_MASK_SCALE_FACTOR_SHIFT                         16  /*  2^16 */
#define IPL_PHI_TABLE_SCALE_FACTOR_SHIFT                     8  /*   2^8 */
#define IPL_PHI_TABLE_SCALE_FACTOR                         256  /*   2^8 */
#define IPL_BINARIZE_MIN                                  0x00
#define IPL_BINARIZE_MAX                                  0xFF
#define IPL_GRADIENT_WORD_NAN            (uint32_t)0x12121212u
#define IPL_GRADIENT_BYTE_NAN                             0x12   /*   18 */
#define IPL_GRADIENT_SCALE_FACTOR                         0x40   /*   64 */

/*****************************************************************************
 * Declaration of module wide TYPEs
 *****************************************************************************/

/*****************************************************************************
 * Definition of module wide (CONST-) CONSTANTs
 *****************************************************************************/

/* phi orientation table based on the order of ArcTanTable
```

```
   i.e.: 90 deg -> 80 deg
*/
const static uint8_t PhiTable[18] =
    {9,10,11,12,13,14,15,16,17,0,1,2,3,4,5,6,7,8};

/*****************************************************************************
 * Code of module wide FUNCTIONS
 *****************************************************************************/

/*****************************************************************************/
/**
 * \brief   Search the maximum value in a the Hough accumulator pu8HoughAcc, this
 *          value and its x and y coordinates are passed back to the caller as a
 *          reference
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8HoughAcc     pointer to Hough accumulator
 *          uint16_t    u16rows         rows in pu8HoughAcc
 *          uint16_t    u16cols         cols in pu8HoughAcc
 *          uint8_t *   pu8HoughMax     pointer to max value in Hough accumulator
 *          uint16_t *  pu16HoughMax_x  x coordinate of max value in Hough accumulator
 *          uint16_t *  pu16HoughMax_y  y coordinate of max value in Hough accumulator
 *
 * \return  void
 */
void hough_space_max_search (
        const uint8_t *    pu8HoughAcc,
        uint16_t           u16rows,
        uint16_t           u16cols,
        uint8_t *          pu8HoughMax,
        uint16_t *         pu16HoughMax_x,
        uint16_t *         pu16HoughMax_y)
{
    uint32_t i_index;
    uint32_t byte_count  = (u16rows * u16cols);
    uint16_t ha_x_index  = 0;
    uint16_t ha_y_index  = 0;

    /* initialize max values */
    *pu8HoughMax    = 0;
```

```
    *pu16HoughMax_x = 0;
    *pu16HoughMax_y = 0;

    /* init cycle counter */
    IPL_MEASURE_RESET;

    /* Traverse byte by byte the Hough accumulator searching for a max */
    for (i_index = 0; i_index < byte_count; i_index++, pu8HoughAcc++) {

        /* if current Hough space max value is greater, update */
        if (*pu8HoughAcc > *pu8HoughMax) {

            *pu8HoughMax    = *pu8HoughAcc;
            *pu16HoughMax_x = ha_x_index;
            *pu16HoughMax_y = ha_y_index;
        }

        /* adjust accumulator x index */
        ha_x_index++;
        if (ha_x_index >= u16cols) {
            /* reset accumulator x index */
            ha_x_index = 0;

            /* adjust accumulator y index */
            ha_y_index += 1;
        }
    }

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
/**
 * \brief   Set a zero in all Hough accumulator's bins
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8HoughAcc     pointer to Hough accumulator
```

```
 *          uint32_t    u32ImageSize    size of Hough accumulator
 *
 * \return  void
 */
void clear_accumulator (
        uint8_t *   pu8HoughAcc,
        uint32_t    u32ImageSize)
{
    uint32_t i_index;
    uint32_t word_count  = u32ImageSize / 4;
    uint32_t long_count  = word_count / 2;

    uint32_t * pu32HoughAcc_1 = (uint32_t *) pu8HoughAcc;
    uint32_t * pu32HoughAcc_2 = (uint32_t *)(pu8HoughAcc + 4);

    /* init cycle counter */
    IPL_MEASURE_RESET;

    /* Loop for Horizontal lines */
    for (i_index = 0; i_index < long_count; i_index++,
                                            pu32HoughAcc_1 += 2,
                                            pu32HoughAcc_2 += 2) {

        /* clear two words at a time */
        *pu32HoughAcc_1 = 0x00000000;
        *pu32HoughAcc_2 = 0x00000000;
    }

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
/**
 * \brief   Performs Hough-space transform based on edge gradient information
 *
 *          This function traverses the Hough-space matrix by double word, word
 *          and then byte by byte looking for directional information.
 *          When directional information is found, the number of phi elements
```

```
 *          for the same orientation is obtained from pu16PhiLength.
 *          Then the template values r, sin(alpha) and cos(alpha) are loaded
 *          for the same orientation.
 *          Hough-space indexes are calculated and Hough accumlator gets
 *          a vote for this point.
 *
 *          Note: alpha sine and cosine tables are assumed to be scaled by 2^8 (256)
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *  pu8PhiImage      pointer to input image represented with
gradient orientation info
 *          uint16_t   u16rows          pu8PhiImage rows
 *          uint16_t   u16cols          pu8PhiImage cols
 *          uint16_t * pu16PhiLength    pointer to input array that describes number
of phi-sorted template elements
 *          uint16_t * pu16PhiIndex     pointer to input array that provides indexes
of phi-matching template elements
 *          int16_t *  pafAlphaSine     pointer to input array with pre-calculated
template values of sin(alpha)
 *          int16_t *  pafAlphaCos      pointer to input array with pre-calculated
template values of cos(alpha)
 *          int16_t *  pafR             pointer to input array with pre-calculated
template values of r
 *          uint8_t *  pu8HoughAcc      pointer to output image of Hough-space
accumulator
 *
 * \return  void
 */
void hough_transform_accumulate (
        const uint8_t * pu8PhiImage,
        uint16_t        u16rows,
        uint16_t        u16cols,
        uint16_t *      pu16PhiLength,
        uint16_t *      pu16PhiIndex,
        int16_t *       pafAlphaSine,
        int16_t *       pafAlphaCos,
        int16_t *       pafR,
        uint8_t *       pu8HoughAcc)
{
    uint32_t i_index;
```

```
uint32_t j_index;
uint32_t k_index;
uint32_t byte_count  = (u16rows * u16cols);
uint32_t word_count  = byte_count / 4;
uint32_t long_count  = word_count / 2;
uint32_t bytes_cycle = 4;

uint32_t * pu32Phi_1 = (uint32_t *) pu8PhiImage;
uint32_t * pu32Phi_2 = (uint32_t *)(pu8PhiImage + 4);

uint16_t im_x_index  = 0;
uint16_t im_y_index  = 0;

uint16_t TempLength  = 0;
uint16_t TempIndex   = 0;
uint16_t phi_index   = 0;
int32_t  r_value     = 0;
int32_t  alpha_sine  = 0;
int32_t  alpha_cos   = 0;
int32_t  hough_x_idx = 0;
int32_t  hough_y_idx = 0;
uint32_t hough_space_idx = 0;
uint8_t  hough_value = 0;

/* init cycle counter */
IPL_MEASURE_RESET;

for (i_index = 0; i_index < long_count; i_index++,
                                        pu32Phi_1 += 2,
                                        pu32Phi_2 += 2) {

    /* if the first word is equal to the reference update the indexes */
    if (*pu32Phi_1 >= IPL_GRADIENT_WORD_NAN) {

        /* if first and second words are equal */
        if (*pu32Phi_1 == *pu32Phi_2) {

            /* two words have been analyzed
             * adjust the indexes and repeat words loop */
            pu8PhiImage += 8;
            im_x_index  += 8;
```

```
            if (im_x_index >= u16cols) {
                /* reset image x index */
                im_x_index = im_x_index - u16cols;

                /* adjust image y index */
                im_y_index += 1;
            }
            continue;
        }

        /* one word has been analyzed
         * adjust image x index */
        pu8PhiImage += 4;
        im_x_index  += 4;

        if (im_x_index >= u16cols) {
            /* reset image x index */
            im_x_index = im_x_index - u16cols;

            /* adjust image y index */
            im_y_index += 1;
        }

        /* iterate byte by byte for one word only */
        bytes_cycle = 4;

    } else {
        /* iterate byte by byte for two words */
        bytes_cycle = 8;
    }

    /* load byte by byte */
    for (j_index = 0; j_index < bytes_cycle; j_index++, pu8PhiImage++) {

        /* if phi orientation is NaN, don't accumulate */
        if (*pu8PhiImage >= IPL_GRADIENT_BYTE_NAN) {
            goto skip_acc;
        }

        /* get the number of phi elements for this orientation */
```

```
            TempLength = pu16PhiLength[*pu8PhiImage];

            /* If template does not have matching phi angle,
             * continue with next point */
            if (TempLength == 0) {
                goto skip_acc;
            }

            /* load all template indexes that match same phi
             * angle of current pixel */
            for (k_index = 0, phi_index = *pu8PhiImage;
                    k_index < TempLength;
                    k_index++, phi_index += 18)
            {
                /* get the index and adjust for zero index */
                TempIndex = pu16PhiIndex[phi_index] - 1;

                /* Load corresponding r value */
                r_value = pafR[TempIndex];

                /* Load corresponding alpha sine value */
                alpha_sine = pafAlphaSine[TempIndex];

                /* Load corresponding alpha cosine value */
                alpha_cos = pafAlphaCos[TempIndex];

                /*-------------------------------------------------------------
                 * Hough-space mapping (x-coordinate)
                 *      Hough_x_index = Im_x_index - r*cos(alpha)
                 *
                 * Hough-space mapping (y-coordinate)
                 *      Hough_y_index = Im_y_index - r*sin(alpha)
                 *-------------------------------------------------------------*/

                /* alpha sine and cosine tables are scaled by 2^8 */
                hough_y_idx = (int32_t)(im_y_index - ((r_value * alpha_sine) /
IPL_PHI_TABLE_SCALE_FACTOR));
                hough_x_idx = (int32_t)(im_x_index - ((r_value * alpha_cos)  /
IPL_PHI_TABLE_SCALE_FACTOR));

                /* make sure resulting X and Y indexes fit within
```

```
                 * accumulator dimensions else, do not accumulate */
                if ((hough_y_idx >= 0) && (hough_y_idx < (int32_t)u16rows) &&
                    (hough_x_idx >= 0) && (hough_x_idx < (int32_t)u16cols))
                {
                    /* ----------------- Hough accumulate ------------------- */

                    /* calculate absolute index into Hough space */
                    hough_space_idx = (uint32_t)((hough_y_idx * u16cols) +
hough_x_idx);

                    /* get current value at Hough space location */
                    hough_value = pu8HoughAcc[hough_space_idx];

                    /* increment accumulator value at location */
                    hough_value++;

                    /* store accumualor value into Hough space */
                    pu8HoughAcc[hough_space_idx] = hough_value;
                }
            }

skip_acc:
            /* adjust image x index */
            im_x_index++;
            if (im_x_index >= u16cols) {
                /* reset image x index */
                im_x_index = im_x_index - u16cols;

                /* adjust image y index */
                im_y_index += 1;
            }
        }
    }

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
```

```
/**
 * \brief   Deletes redundant information about horizontal and vertical
 *          straight lines
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8Image        pointer to image to be processed
 *          uint16_t    u16rows         rows in pu8Image
 *          uint16_t    u16cols         cols in pu8Image
 *
 * \return  void
 */
void image_delete (
        uint8_t *   pu8Image,
        uint16_t    u16rows,
        uint16_t    u16cols)
{
    uint32_t i_index;
    uint32_t byte_count  = (u16rows * u16cols);
    uint32_t word_count  = byte_count / 4;
    uint32_t long_count  = word_count / 2;

    uint32_t * pu32Image_1 = (uint32_t *) pu8Image;
    uint32_t * pu32Image_2 = (uint32_t *)(pu8Image + 4);

    /* init cycle counter */
    IPL_MEASURE_RESET;

    /* Loop for Horizontal lines */
    for (i_index = 0; i_index < long_count; i_index++,
                                            pu32Image_1 += 2,
                                            pu32Image_2 += 2) {

        /* load words from gradien images X and Y and look for blank pixels */
        if (*pu32Image_1 == 0x00000000u) {
            *pu32Image_1 = IPL_GRADIENT_BYTE_NAN;
        }

        if (*pu32Image_2 == 0x00000000u) {
            *pu32Image_2 = IPL_GRADIENT_BYTE_NAN;
        }
```

```
    }

    /* Loop for Vertical lines */
    for (i_index = 0; i_index < byte_count;) {

        /* Compare byte by byte for vertical orientation */
        if ((9 == pu8Image[i_index]) &&
            (pu8Image[i_index]                == pu8Image[i_index + u16cols]) &&
            (pu8Image[i_index + u16cols]      == pu8Image[i_index + (u16cols * 2)])
&&
            (pu8Image[i_index + (u16cols * 2)] == pu8Image[i_index + (u16cols * 3)])
&&
            (pu8Image[i_index + (u16cols * 3)] == pu8Image[i_index + (u16cols * 4)]))
        {
            pu8Image[i_index + (u16cols * 4)] = IPL_GRADIENT_BYTE_NAN;
        }

        /* Increase index and check for when when a row is done to
         * jump to the next row, three below... */
        i_index++;
        if ((i_index % u16cols) == 0)
            i_index += u16cols * 3;
    }

#ifndef IPL_COPY_PIXELS_FROM_ORIGINAL_IMG
    {
        uint32_t last_row_offset      = (u16rows * u16cols) - u16cols;
        uint32_t prev_last_row_offset = (u16rows * u16cols) - (2 * u16cols);

        /* Delete first and last two columns */
        for (i_index = 0; i_index < u16rows; i_index++) {
            pu8Image[(u16cols * i_index)]     = IPL_GRADIENT_BYTE_NAN;
            pu8Image[(u16cols * i_index) + 1] = IPL_GRADIENT_BYTE_NAN;
            pu8Image[(u16cols + (u16cols * i_index)) - 1] = IPL_GRADIENT_BYTE_NAN;
            pu8Image[(u16cols + (u16cols * i_index)) - 2] = IPL_GRADIENT_BYTE_NAN;
        }

        /* Delete first and last two rows */
        for (i_index = 0; i_index < u16cols; i_index++) {
            pu8Image[i_index]           = IPL_GRADIENT_BYTE_NAN;
            pu8Image[i_index + u16rows] = IPL_GRADIENT_BYTE_NAN;
```

```
            pu8Image[i_index + last_row_offset]      = IPL_GRADIENT_BYTE_NAN;
            pu8Image[i_index + prev_last_row_offset] = IPL_GRADIENT_BYTE_NAN;
        }
    }
#endif

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
/**
 * \brief   Compute directional information based on a discrete calculation of
 *          the tangent of theta. The resulting gradient orientation image is
 *          stored into pu8PhiImage.
 *
 *          Note: array of discrete values of tan(theta) is assumed to be scaled
 *          by 2^6 (64)
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8GradY        pointer to gradient Y information of an image
 *          uint8_t *   pu8GradX        pointer to gradient X information of an image
 *          uint8_t *   pu8PhiImage     pointer to the resulting gradient orientation
image
 *          uint32_t    u32ImageSize    size of pu8PhiImage
 *          int16_t *   p16ArcTanTable  array of discrete values of tan(theta)
 *
 * \return  void
 */
void image_gradient_orientation (
        const uint8_t *    pu8GradY,
        const uint8_t *    pu8GradX,
        uint8_t *          pu8PhiImage,
        uint32_t           u32ImageSize,
        int16_t *          p16ArcTanTable)
{
    uint32_t i_index;
    uint32_t j_index;
```

```
uint32_t k_index;
uint32_t word_count  = u32ImageSize / 4;
int16_t  division    = 0;

uint32_t * pu32GradY    = (uint32_t *)pu8GradY;
uint32_t * pu32GradX    = (uint32_t *)pu8GradX;
uint32_t * pu32PhiImage = (uint32_t *)pu8PhiImage;

/* init cycle counter */
IPL_MEASURE_RESET;

for (i_index = 0; i_index < word_count; i_index++,
                                        pu32GradY++,
                                        pu32GradX++,
                                        pu32PhiImage++) {

    /* load words from gradien images X and Y and look for blank pixels */
    if (*pu32GradY == 0x00000000u) {
        if (*pu32GradX == 0x00000000u) {
            /* empty words, set NaN word in gradient image,
             * increase image indexes and continue */
            *pu32PhiImage = IPL_GRADIENT_WORD_NAN;

            pu8GradY    += 4;
            pu8GradX    += 4;
            pu8PhiImage += 4;
            continue;
        }
    }

    /* words are not empty, start analyzing byte by byte for the
     * current word */
    for (j_index = 0; j_index < 4; j_index++,
                                   pu8GradY++,
                                   pu8GradX++,
                                   pu8PhiImage++) {

        if ((*pu8GradY == 0) && (*pu8GradX == 0)) {
            /* blank pixel */
            *pu8PhiImage = IPL_GRADIENT_BYTE_NAN;
            continue;
```

```
            }

            /* if divisor is zero, phi = 90 deg */
            if (*pu8GradX == 0) {
                *pu8PhiImage = PhiTable[0];
                continue;
            }

            /* divide gradY/gradX */
            division = ((int8_t)*pu8GradY * IPL_GRADIENT_SCALE_FACTOR) /
(int8_t)*pu8GradX;

            /* traverse the arctan table looking for the matching gradient */
            for (k_index = 0; k_index < 18; k_index++) {

                if (division <= p16ArcTanTable[k_index]) {

                    /* set gradient direction for the
                     * corresponding angle */
                    *pu8PhiImage = PhiTable[k_index];
                    break;
                }

                if (k_index == 17) {
                    /* greater than 85° - set 90 deg gradient value */
                    *pu8PhiImage = PhiTable[0];
                }
            }
        }
    }

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
/**
 * \brief   This function merges dual-edge borders generated by using directional
 *          gradient masks, as in Ando's directional masks into single-edge borders.
```

```
 *          The resulting single-edge border images are stored into the same
 *          input images pu8GradY and pu8GradX.
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8GradY        pointer to gradient Y information of an image
 *          uint8_t *   pu8GradX        pointer to gradient X information of an image
 *          uint16_t    u16rows         rows in pu8GradY
 *          uint16_t    u16cols         cols in pu8GradY
 *
 * \return  void
 */
void image_edge_thinning (
        uint8_t *        pu8GradY,
        uint8_t *        pu8GradX,
        uint16_t         u16rows,
        uint16_t         u16cols)
{
    uint32_t i_index;
    uint32_t j_index;
    uint32_t pixel_count = u16rows * u16cols;
    uint32_t word_count  = pixel_count / 4;

    uint32_t * pu32GradY = (uint32_t *)pu8GradY;
    uint32_t * pu32GradX = (uint32_t *)pu8GradX;

    /* init cycle counter */
    IPL_MEASURE_RESET;

    for (i_index = 0; i_index < word_count; i_index++, pu32GradY++, pu32GradX++) {

        /* load words from gradien images X and Y */
        if (*pu32GradY == 0x00000000u) {
            if (*pu32GradX == 0x00000000u) {
                /* empty words, increase image indexes and continue */
                pu8GradY += 4;
                pu8GradX += 4;
                continue;
            }
        }
```

```
/* words are not empty, start analyzing byte by byte for the
 * current word */
for (j_index = 0; j_index < 4; j_index++, pu8GradY++, pu8GradX++) {

    if ((*pu8GradY == 128) || (*pu8GradX == 128)) {
        /* blank pixel */
        goto pixel_blank;
    }

    if (*pu8GradY + *pu8GradX == 255) {
        /* complement */
        if (*pu8GradY < 186)
            goto pixel_blank;
        continue;
    }

    if (*pu8GradY == *pu8GradX) {
        /* same */
        if (*pu8GradY < 69)
            continue;
        goto pixel_blank;
    }

    /* compare with patterns */
    if (*pu8GradY > 35) {
        /* compare with straigh line */
        if (*pu8GradY > 125) {
            /* compare high */
            if (*pu8GradY < 128) {
                continue;
            }

            if (*pu8GradY > 220) {
                continue;
            }
        }

        /* compare X */
        if (*pu8GradX > 35) {
            /* compare with straigh line */
            if (*pu8GradX > 125) {
```

```
                        /* compare high */
                        if (*pu8GradY < 128) {
                            continue;
                        }

                        if (*pu8GradY < 220) {
                            goto pixel_blank;
                        }

                        continue;
                    }

                    goto pixel_blank;
                }
            }

            continue;

pixel_blank:
            /* set blank pixels */
            *pu8GradY = 0;
            *pu8GradX = 0;
        }
    }

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
/**
 * \brief   Subtracts each element in pu8ImageA from the corresponding element
 *          in pu8ImageB and returns the difference in the corresponding element
 *          of the output pu8ImageResult
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8ImageA       pointer to input image A
 *          uint8_t *   pu8ImageB       pointer to input image B
```

```
 *          uint16_t    u16rows         rows in pu8ImageA
 *          uint16_t    u16cols         cols in pu8ImageA
 *          uint8_t *   pu8ImageResult  pointer to result image = A - B
 *
 * \return  void
 */
void image_subtract (
        const uint8_t *  pu8ImageA,
        const uint8_t *  pu8ImageB,
        uint16_t         u16rows,
        uint16_t         u16cols,
        uint8_t *        pu8ImageResult)
{
    uint32_t i_index;
    uint32_t pixel_count = u16rows * u16cols;

    /* init cycle counter */
    IPL_MEASURE_RESET;

    for (i_index = 0; i_index < pixel_count; i_index++,
                                             pu8ImageResult++,
                                             pu8ImageA++,
                                             pu8ImageB++)
    {
        *pu8ImageResult = *pu8ImageA - *pu8ImageB;
    }

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
/**
 * \brief   Converts the grayscale image pu8Image to a binary image. The output
 *          image pu8OutImage replaces all pixels in the input image grater than
 *          u8Threshold with the value IPL_BINARIZE_MAX and replaces all other
 *          pixels with the value IPL_BINARIZE_MIN
 *
 * \author  Roberto Ortega
```

```
 *
 * \param   uint8_t *   pu8Image        pointer to input image
 *          uint16_t    u16rows         pu8Image rows
 *          uint16_t    u16cols         pu8Image cols
 *          uint8_t     u8Threshold     threshold value
 *          uint8_t *   pu8OutImage     pointer to output filtered image
 *
 * \return  void
 */
void image_binarize (
        const uint8_t *  pu8Image,
        uint16_t         u16rows,
        uint16_t         u16cols,
        uint8_t          u8Threshold,
        uint8_t *        pu8OutImage)
{
    uint32_t i_index;
    uint32_t pixel_count = u16rows * u16cols;

    /* init cycle counter */
    IPL_MEASURE_RESET;

    for (i_index = 0; i_index < pixel_count; i_index++)
    {
        if (*pu8Image <= u8Threshold) {
            *pu8OutImage = IPL_BINARIZE_MIN;
        } else {
            *pu8OutImage = IPL_BINARIZE_MAX;
        }

        pu8Image++;
        pu8OutImage++;
    }

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
```

```
/**
 * \brief   Filtering of a u16rows x u16cols Image using a 3x3 mask based on
 *          spatial correlation. The result is stored into pu8OutImage.
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8Image        pointer to input image
 *          uint16_t    u16rows         pu8Image rows
 *          uint16_t    u16cols         pu8Image cols
 *          float *     pfMask          pointer to filter's mask
 *          uint8_t *   pu8OutImage     pointer to output filtered image
 *
 * \return  void
 */
void spatial_filt_int_3x3 (
        const uint8_t *  pu8Image,
        uint16_t         u16rows,
        uint16_t         u16cols,
        float *          pfMask,
        uint8_t *        pu8OutImage)
{
    uint32_t Filtered3x3scaled;
    int32_t  Mask3x3Scaled[3][3];
    uint16_t i_index;
    uint16_t j_index;

    /* Convert to integer and scale up correlation mask in order to avoid
     * loosing resolution */
    for (i_index = 0; i_index < 3; i_index++)
    {
        for (j_index = 0; j_index < 3; j_index++)
        {
            /* Mask to be scaled up by a factor of 2^16*/
            Mask3x3Scaled[i_index][j_index] = (int32_t)((*pfMask++) *
IPL_MASK_SCALE_FACTOR);
        }
    }

    /* init cycle counter */
    IPL_MEASURE_RESET;
```

```
#ifdef IPL_COPY_PIXELS_FROM_ORIGINAL_IMG
    /* keep first row pixels as original image */
    for (i_index = 0; i_index <= (u16cols - 1); i_index++)
    {
        *pu8OutImage++ = *pu8Image++;
    }
#else
    /* set image index to row i + 1 */
    pu8OutImage += u16cols;
    pu8Image += u16cols;
#endif

    /* Perform correlation operation */
    for (i_index = 0; i_index <= (u16rows - 3); i_index++)
    {
#ifdef IPL_COPY_PIXELS_FROM_ORIGINAL_IMG
        /* set first pixel - keep first col pixels as original image */
        *pu8OutImage++ = *pu8Image;
#else
        /* get next filtered image index */
        pu8OutImage++;
#endif

        for (j_index = 0; j_index <= (u16cols - 3); j_index++)
        {
            Filtered3x3scaled = (uint32_t)(
                    (*(pu8Image - u16cols)       * Mask3x3Scaled[0][0]) +
                    (*(pu8Image - u16cols + 1)   * Mask3x3Scaled[0][1]) +
                    (*(pu8Image - u16cols + 2)   * Mask3x3Scaled[0][2]) +
                    (*(pu8Image)                 * Mask3x3Scaled[1][0]) +
                    (*(pu8Image + 1)             * Mask3x3Scaled[1][1]) +
                    (*(pu8Image + 2)             * Mask3x3Scaled[1][2]) +
                    (*(pu8Image + u16cols)       * Mask3x3Scaled[2][0]) +
                    (*(pu8Image + u16cols + 1)   * Mask3x3Scaled[2][1]) +
                    (*(pu8Image + u16cols + 2)   * Mask3x3Scaled[2][2]));

            /* Scale down result */
            *pu8OutImage++ = (uint8_t)(Filtered3x3scaled >>
IPL_MASK_SCALE_FACTOR_SHIFT);

            /* get next image pixel */
```

```
            pu8Image++;
        }

#ifdef IPL_COPY_PIXELS_FROM_ORIGINAL_IMG
        /* get next image pixel */
        pu8Image++;

        /* keep last col pixels as original image */
        *pu8OutImage++ = *pu8Image++;
#else
        /* get next image pixel */
        pu8Image += 2;

        /* set filtered image pixel position to next row */
        pu8OutImage++;
#endif
    }

#ifdef IPL_COPY_PIXELS_FROM_ORIGINAL_IMG
    /* keep last row pixels as original image */
    for (i_index = 0; i_index <= (u16cols - 1); i_index++)
    {
        *pu8OutImage++ = *pu8Image++;
    }
#endif

    /* get clock cycle count */
    IPL_MEASURE_GET;

    return;
}

/*****************************************************************************/
/**
 * \brief   Filtering of a u16rows x u16cols Image using a 2x2 mask based on
 *          spatial correlation. The result is stored into pu8OutImage.
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8Image        pointer to input image
 *          uint16_t    u16rows         pu8Image rows
```

```
 *          uint16_t    u16cols         pu8Image cols
 *          float *     pfMask          pointer to filter's mask
 *          uint8_t *   pu8OutImage     pointer to output filtered image
 *
 * \return  void
 */
void spatial_filt_int_2x2 (
        const uint8_t *  pu8Image,
        uint16_t         u16rows,
        uint16_t         u16cols,
        float *          pfMask,
        uint8_t *        pu8OutImage)
{
    /* Intermediate scaled up image - temporary pixel calculation */
    uint32_t Filtered2x2scaled;
    /* Intermediate Mask in integer numbers to accelerate execution */
    int32_t  Mask2x2Scaled[2][2];
    uint16_t i_index;
    uint16_t j_index;

    /* Convert to integer and scale up correlation mask in order to avoid
     * loosing resolution */
    for (i_index = 0; i_index < 2; i_index++)
    {
        for (j_index = 0; j_index < 2; j_index++)
        {
            /* Mask to be scaled up by a factor of 2^16*/
            Mask2x2Scaled[i_index][j_index] = (int32_t)((*pfMask++) *
IPL_MASK_SCALE_FACTOR);
        }
    }

    /* init cycle counter */
    IPL_MEASURE_RESET;

#ifdef IPL_COPY_PIXELS_FROM_ORIGINAL_IMG
    /* set first pixel - first column is not calculated */
    *pu8OutImage = *pu8Image;
#endif

    /* Perform correlation operation */
```

```
    for (i_index = 0; i_index <= (u16rows - 2); i_index++)
    {
        /* get next filtered image index */
        pu8OutImage++;

        for (j_index = 0; j_index <= (u16cols - 2); j_index++)
        {
            Filtered2x2scaled = (uint32_t)(
                    (*(pu8Image)               * Mask2x2Scaled[0][0]) +
                    (*(pu8Image + u16cols)     * Mask2x2Scaled[1][0]) +
                    (*(pu8Image + 1)           * Mask2x2Scaled[0][1]) +
                    (*(pu8Image + 1 + u16cols) * Mask2x2Scaled[1][1]));

            /* Scale down result */
            *pu8OutImage++ = (uint8_t)(Filtered2x2scaled >>
IPL_MASK_SCALE_FACTOR_SHIFT);

            /* get next image pixel */
            pu8Image++;
        }

#ifdef IPL_COPY_PIXELS_FROM_ORIGINAL_IMG
        /* get next image pixel from original image */
        *pu8OutImage = *++pu8Image;
#else
        /* get next image pixel - first column is not calculated */
        pu8Image++;
#endif
    }

#ifdef IPL_COPY_PIXELS_FROM_ORIGINAL_IMG
    /* keep last row pixels as original image */
    for (i_index = 0; i_index <= (u16cols - 2); i_index++)
    {
        *++pu8OutImage = *++pu8Image;
    }
#endif

    /* get clock cycle count */
    IPL_MEASURE_GET;
```

```
    return;
}

/*****************************************************************************/
```

B. IMAGE PROCESSING LIBRARY HEADER

```
/*****************************************************************************/
/**
  \file       ipl.h
  \brief      Image processing library header file
  \author     Roberto Ortega
              Based on the ASM implementation by Abraham Tezmol
  \version    1.0
  \date       2/January/2017
  */
/*****************************************************************************/

#ifdef  __IPL_H_
#define __IPL_H_ extern
#endif

#ifndef __IPL_H_
#define __IPL_H_

/* custom type definitions */
#include "typedefs.h"

/* This variable stores the clock cycle count of the last IPL API call if
 *  the compiler switch IPL_MEASURE_CLK_CYCLES is enabled.
*/
extern uint32_t ipl_clk_cycles;

/*~~~~~~~~~~~~~~~~ Image Processing Library definitions ~~~~~~~~~~~~~~~~~~~~*/

/* Discrete table for tan(theta) -- scaled by 2^6 (64) -- 90 deg -> 80 deg */
#pragma alignvar(4)
const __IPL_H_ INT16 ArcTanTable[1][18] =
{-732,-239,-137,-91,-64,-45,-30,-17,-6,6,17,30,45,64,91,137,239,732};

/* Gaussian mask 3x3 for spatial filter */
#pragma alignvar(4)
__IPL_H_ float GaussianMask_3x3[3][3] =
{
    0.1088,    0.1123,    0.1088,
```

```
    0.1123,    0.1158,    0.1123,
    0.1088,    0.1123,    0.1088
};

/* Average mask 3x3 for spatial filter */
#pragma alignvar(4)
__IPL_H_ float AvgMask_3x3[3][3] =
{
    0.11111,    0.11111,    0.11111,
    0.11111,    0.11111,    0.11111,
    0.11111,    0.11111,    0.11111
};

/* Average mask 2x2 for spatial filter */
#pragma alignvar(4)
__IPL_H_ float AvgMask_2x2[2][2] =
{
    0.2500,     0.2500,
    0.2500,     0.2500
};

/* Ando directional Y mask 3x3 for spatial filter */
#pragma alignvar(4)
__IPL_H_ float AndoMaskY_3x3[3][3]=
{
    -0.112737,  0.000000,   0.112737,
    -0.274526,  0.000000,   0.274526,
    -0.112737,  0.000000,   0.112737
};

/* Ando directional X mask 3x3 forspatial filter */
#pragma alignvar(4)
__IPL_H_ float AndoMaskX_3x3[3][3]=
{
    -0.112737,  -0.274526,  -0.112737,
     0.000000,   0.000000,   0.000000,
     0.112737,   0.274526,   0.112737
};

/*~~~~~~~~~~~~ Image Processing Library API Definition ~~~~~~~~~~~~~~~~~~~~~~~~~*/
```

B. IMAGE PROCESSING LIBRARY HEADER

```
/*****************************************************************************/
/**
 * \brief   Filtering of a u16rows x u16cols Image using a 3x3 mask based on
 *          spatial correlation. The result is stored into pu8OutImage.
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8Image        pointer to input image
 *          uint16_t    u16rows         pu8Image rows
 *          uint16_t    u16cols         pu8Image cols
 *          float *     pfMask          pointer to filter's mask
 *          uint8_t *   pu8OutImage     pointer to output filtered image
 *
 * \return  void
 *****************************************************************************/
void        spatial_filt_int_3x3        (const uint8_t *    pu8Image,
                                         uint16_t           u16rows,
                                         uint16_t           u16cols,
                                         float *            pfMask,
                                         uint8_t *          pu8OutImage);

/*****************************************************************************/
/**
 * \brief   Filtering of a u16rows x u16cols Image using a 2x2 mask based on
 *          spatial correlation. The result is stored into pu8OutImage.
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8Image        pointer to input image
 *          uint16_t    u16rows         pu8Image rows
 *          uint16_t    u16cols         pu8Image cols
 *          float *     pfMask          pointer to filter's mask
 *          uint8_t *   pu8OutImage     pointer to output filtered image
 *
 * \return  void
 *****************************************************************************/
void        spatial_filt_int_2x2        (const uint8_t *    pu8Image,
                                         uint16_t           u16rows,
                                         uint16_t           u16cols,
                                         float *            pfMask,
                                         uint8_t *          pu8OutImage);
```

```
/*****************************************************************************/
/**
 * \brief   Converts the grayscale image pu8Image to a binary image. The output
 *          image pu8OutImage replaces all pixels in the input image grater than
 *          u8Threshold with the value IPL_BINARIZE_MAX and replaces all other
 *          pixels with the value IPL_BINARIZE_MIN
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8Image        pointer to input image
 *          uint16_t    u16rows         pu8Image rows
 *          uint16_t    u16cols         pu8Image cols
 *          uint8_t     u8Threshold     threshold value
 *          uint8_t *   pu8OutImage     pointer to output filtered image
 *
 * \return  void
 *****************************************************************************/
void        image_binarize              (const uint8_t *    pu8Image,
                                         uint16_t           u16rows,
                                         uint16_t           u16cols,
                                         uint8_t            u8Threshold,
                                         uint8_t *          pu8OutImage);

/*****************************************************************************/
/**
 * \brief   Subtracts each element in pu8ImageA from the corresponding element
 *          in pu8ImageB and returns the difference in the corresponding element
 *          of the output pu8ImageResult
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8ImageA       pointer to input image A
 *          uint8_t *   pu8ImageB       pointer to input image B
 *          uint16_t    u16rows         rows in pu8ImageA
 *          uint16_t    u16cols         cols in pu8ImageA
 *          uint8_t *   pu8ImageResult  pointer to result image = A - B
 *
 * \return  void
 *****************************************************************************/
void        image_subtract              (const uint8_t *    pu8ImageA,
```

```
                                        const uint8_t *    pu8ImageB,
                                        uint16_t           u16rows,
                                        uint16_t           u16cols,
                                        uint8_t *          pu8ImageResult);

/*****************************************************************************/
/**
 * \brief   Compute directional information based on a discrete calculation of
 *          the Arc tangent. The resulting gradient orientation image is stored
 *          into pu8PhiImage.
 *
 *          Note: alpha sine and cosine tables are assumed to be scaled by 2^6 (64)
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8GradY        pointer to gradient Y information of an image
 *          uint8_t *   pu8GradX        pointer to gradient X information of an image
 *           uint8_t *   pu8PhiImage     pointer to the resulting gradient orientation
image
 *          uint32_t    u32ImageSize    size of pu8PhiImage
 *          int16_t *   p16ArcTanTable  array of discrete values of tan(theta)
 *
 * \return  void
 *****************************************************************************/
void        image_gradient_orientation  (const uint8_t *    pu8GradY,
                                         const uint8_t *    pu8GradX,
                                         uint8_t *          pu8PhiImage,
                                         uint32_t           u32ImageSize,
                                         int16_t *          pu16ArcTanTable);

/*****************************************************************************/
/**
 * \brief   Deletes redundant information about horizontal and vertical straight lines
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8Image        pointer to image to be processed
 *          uint16_t    u16rows         rows in pu8Image
 *          uint16_t    u16cols         cols in pu8Image
 *
 * \return  void
```

```
 ******************************************************************************/
void        image_delete                (uint8_t *          pu8Image,
                                         uint16_t           u16rows,
                                         uint16_t           u16cols);


/******************************************************************************/
/**
 * \brief   This function merges dual-edge borders generated by using directional
 *          gradient masks, as in Ando's directional masks into single-edge borders.
 *          The resulting single-edge border images are stored into the same
 *          input images pu8GradY and pu8GradX.
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8GradY        pointer to gradient Y information of an image
 *          uint8_t *   pu8GradX        pointer to gradient X information of an image
 *          uint16_t    u16rows         rows in pu8GradY
 *          uint16_t    u16cols         cols in pu8GradY
 *
 * \return  void
 ******************************************************************************/
void        image_edge_thinning         (uint8_t *          pu8GradY,
                                         uint8_t *          pu8GradX,
                                         uint16_t           u16rows,
                                         uint16_t           u16cols);

/******************************************************************************/
/**
 * \brief   Performs Hough-space transform based on edge gradient information
 *
 *          This function traverses the Hough-space matrix by double word, word
 *          and then byte by byte looking for directional information.
 *          When directional information is found, the number of phi elements
 *          for the same orientation is obtained from pu16PhiLength.
 *          Then the template values r, sin(alpha) and cos(alpha) are loaded
 *          for the same orientation.
 *          Hough-space indexes are calculated and Hough accumlator gets
 *          a vote for this point.
 *
 *          Note: alpha sine and cosine tables are assumed to be scaled by 2^8 (256)
```

```
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *  pu8PhiImage      pointer to input image represented with gradient
orientation info
 *          uint16_t   u16rows          pu8PhiImage rows
 *          uint16_t   u16cols          pu8PhiImage cols
 *          uint16_t * pu16PhiLength    pointer to input array that describes number of
phi-sorted template elements
 *          uint16_t * pu16PhiIndex     pointer to input array that provides indexes of
phi-matching template elements
 *           int16_t *  pafAlphaSine      pointer to input array with pre-calculated
template values of sin(alpha)
 *           int16_t *  pafAlphaCos       pointer to input array with pre-calculated
template values of cos(alpha)
 *          int16_t *  pafR              pointer to input array with pre-calculated
template values of r
 *         uint8_t *  pu8HoughAcc     pointer to output image of Hough-space accumulator
 *
 * \return  void
 ******************************************************************************/
void        hough_transform_accumulate  (const uint8_t *    pu8PhiImage,
                                         uint16_t           u16rows,
                                         uint16_t           u16cols,
                                         uint16_t *         pu16PhiLength,
                                         uint16_t *         pu16PhiIndex,
                                         int16_t *          pafAlphaSine,
                                         int16_t *          pafAlphaCos,
                                         int16_t *          pafR,
                                         uint8_t *          pu8HoughAcc);

/******************************************************************************/
/**
 * \brief   Set a zero in all Hough accumulator's bins
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8HoughAcc     pointer to Hough accumulator
 *          uint32_t    u32ImageSize    size of Hough accumulator
 *
 * \return  void
```

```
 ******************************************************************************/
void        clear_accumulator           (uint8_t *          pu8HoughAcc,
                                         uint32_t           u32ImageSize);

/******************************************************************************/
/**
 * \brief   Search the maximum value in a the Hough accumulator pu8HoughAcc, this
 *          value and its x and y coordinates are passed back to the caller as a
 *          reference
 *
 * \author  Roberto Ortega
 *
 * \param   uint8_t *   pu8HoughAcc     pointer to Hough accumulator
 *          uint16_t    u16rows         rows in pu8HoughAcc
 *          uint16_t    u16cols         cols in pu8HoughAcc
 *          uint8_t *   pu8HoughMax     pointer to max value in Hough accumulator
 *          uint16_t *  pu16HoughMax_x  x coordinate of max value in Hough accumulator
 *          uint16_t *  pu16HoughMax_y  y coordinate of max value in Hough accumulator
 *
 * \return  void
 ******************************************************************************/
void        hough_space_max_search      (const uint8_t *    pu8HoughAcc,
                                         uint16_t           u16rows,
                                         uint16_t           u16cols,
                                         uint8_t *          pu8HoughMax,
                                         uint16_t *         pu16HoughMax_x,
                                         uint16_t *         pu16HoughMax_y);

/*============================================================================*/
#endif /* __IPL_H_ */
```