BERKELEY DESIGN TECHNOLOGY, INC.

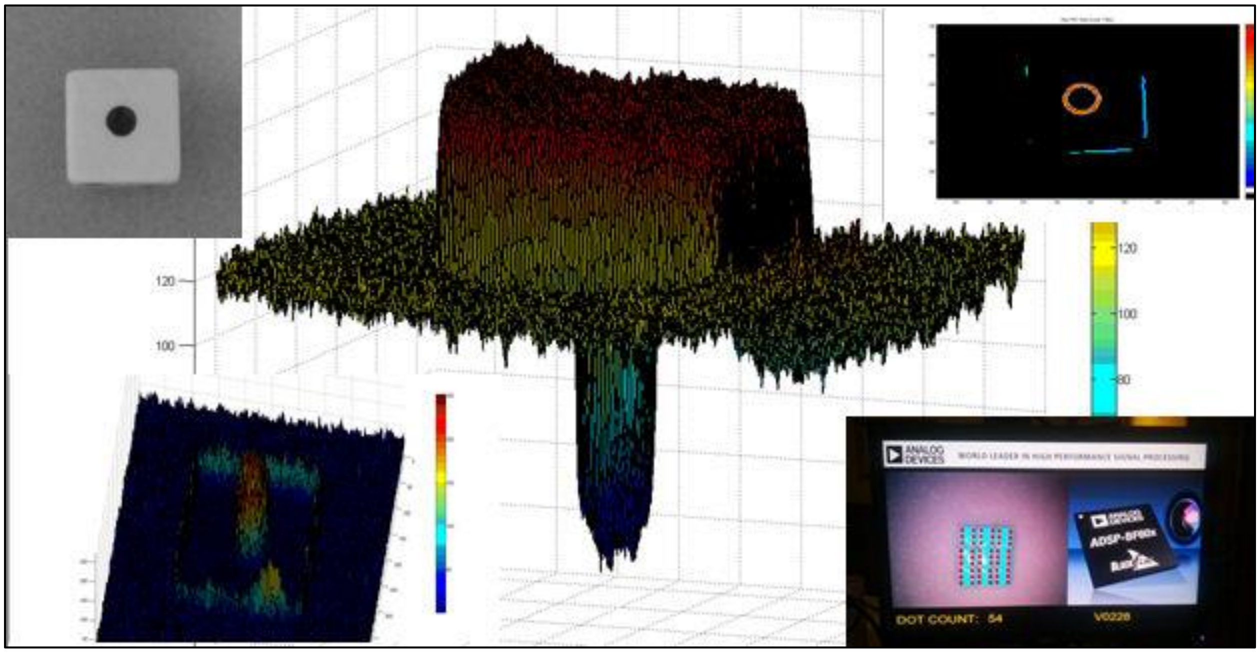
Email: info@BDTI.com
Web: www.BDTI.com

1646 North California Boulevard
Suite 220
Walnut Creek, CA 94596
U.S.A.

Telephone: +1 (925) 954-1411
Fax: +1 (925) 954-1423

BDTI Dice Dot-Counting Demo and Reference Design — Software User's Guide —

+



September 10, 2013

Table of Contents

1. Overview of Demo and Reference Design

The BDTI Dice Dot-Counting Demo and Reference Design (BDTI_DiceDotCountingDemo.zip) demonstrates the computer vision techniques of boundary segmentation and classification on the Analog Devices Blackfin ADSP-BF609 processor using the chip's dual Blackfin CPU/DSP cores and integrated Pipelined Vision Processor (PVP). Boundary segmentation and classification are demonstrated by finding and counting dice dots in real-time using a low-cost chip camera. The dice are thrown within view of the camera and the software automatically finds the dots and counts a total much faster than a human can. This is accomplished under semi-controlled lighting conditions and loosely controlled mechanical setup. Although the problem of counting dice dots sounds simple, factors such as light reflecting off the dice, the need to reject dots on the sides of the dice and irregular dots must be overcome.

The first part of this document covers the boundary segmentation and classification algorithm used by the design to find and count the dice dots. The second part of the document covers implementing the algorithm on the Avnet FinBoard Blackfin ADSP-BF609 development board, part of the Blackfin Embedded Vision Starter Kit. Finally, some example data taken from running the demo is shown and explained. This document is intended for people interested in using the BDTI Dice Dot-Counting Demo and Reference Design as a reference for their own boundary segmentation and classification implementation.

2. About BDTI

The BDTI Dice Dot-Counting Demo and Reference Design was created by BDTI (Berkeley Design Technology, Inc.). BDTI provides product-development engineering services for embedded computer vision and other digital signal processing-based systems. BDTI has expert knowledge of the Analog Devices BF609 and its associated development tools and libraries. BDTI's clients include leading equipment manufacturers and algorithm providers in the consumer, industrial, medical, automotive and mobile device industries. Embedded vision product developers engage BDTI for its deep knowledge of vision applications, algorithms, processors, and tools. Clients describe BDTI as an extraordinarily reliable partner, consistently delivering technically challenging projects on-time, on-budget, and right the first time. For more information about BDTI, please visit www.BDTI.com

3. Algorithm Overview

The heart of the BDTI Dice Dot-Counting Demo and Reference Design classifies the contours into dots and counts the dots. Only contours in the shape of circles are counted as dots.

Figure 1. Algorithm Block Diagram is a block diagram of the dot-counting algorithm. The dot-counting algorithm is made up of layers, as is common with computer vision algorithms. The first layer finds edges in the monochrome frame using a popular edge detector referred to as the Canny edge detector. The edges mark boundaries between different regions in the image. The next layer groups connected edges together into contours. Finally, the contours are tested against a mathematical model of a circle. Circles that fit the mathematical model are counted as dots.

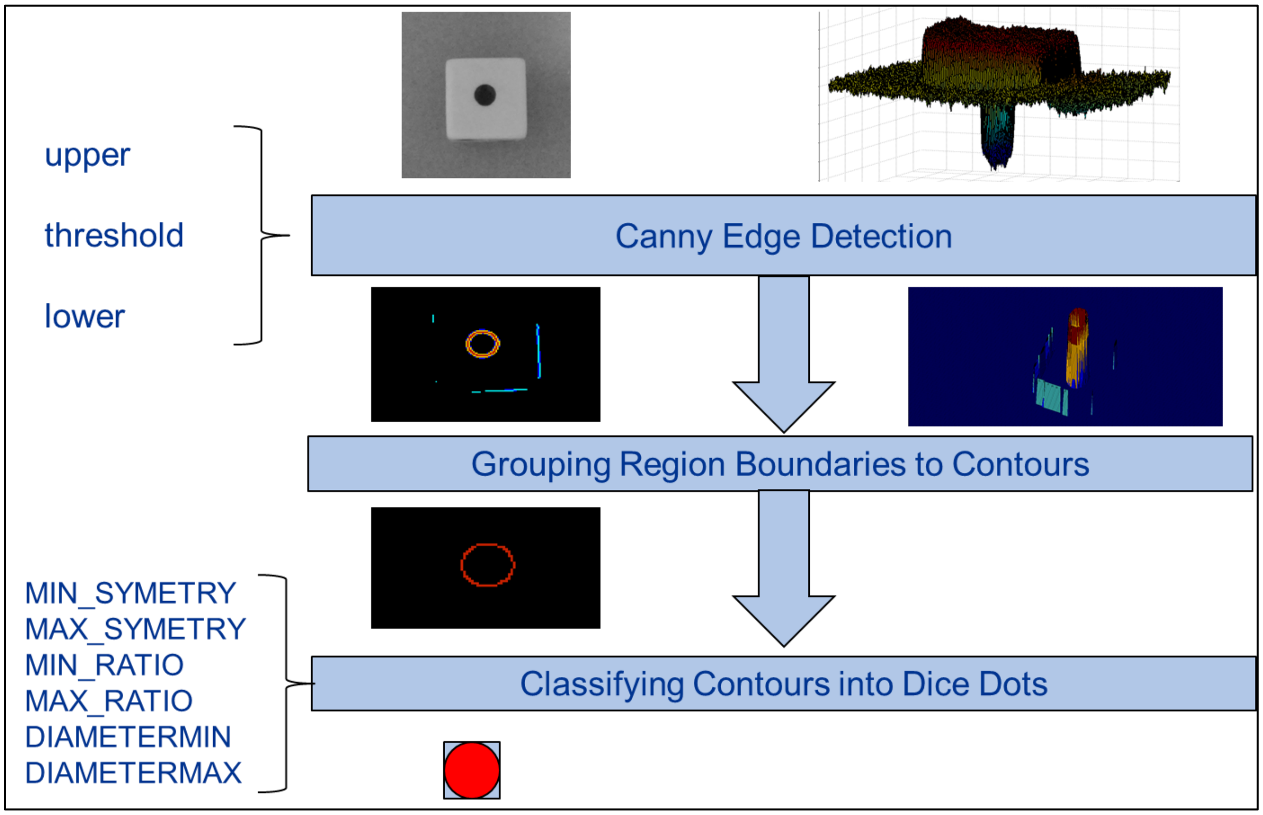


Figure 1. Algorithm Block Diagram

3.1 Finding Region Boundaries Using Canny Edge Detection

"A region usually describes contents (or interior points) that are surrounded by a boundary (or perimeter) which is often called the region's contour." [1] Regions contain pixels of similar value (gray scale). A boundary exists between regions of different gray scale values. The difference of pixel values between regions is an edge, thus edges define boundaries between regions. Figure 2 illustrates the various regions associated with a die. The vertical axis in the 3D plot represents a gray scale value. White is 255, while black is 0. Notice the surface of the die relative to the pink background surface on which the die is sitting in the 3D plot. The dot on the die is especially noticeable in the 3D plot. The goal of this design is to find and count the dice dots. An edge detection algorithm is used to find the edges surrounding the dot.

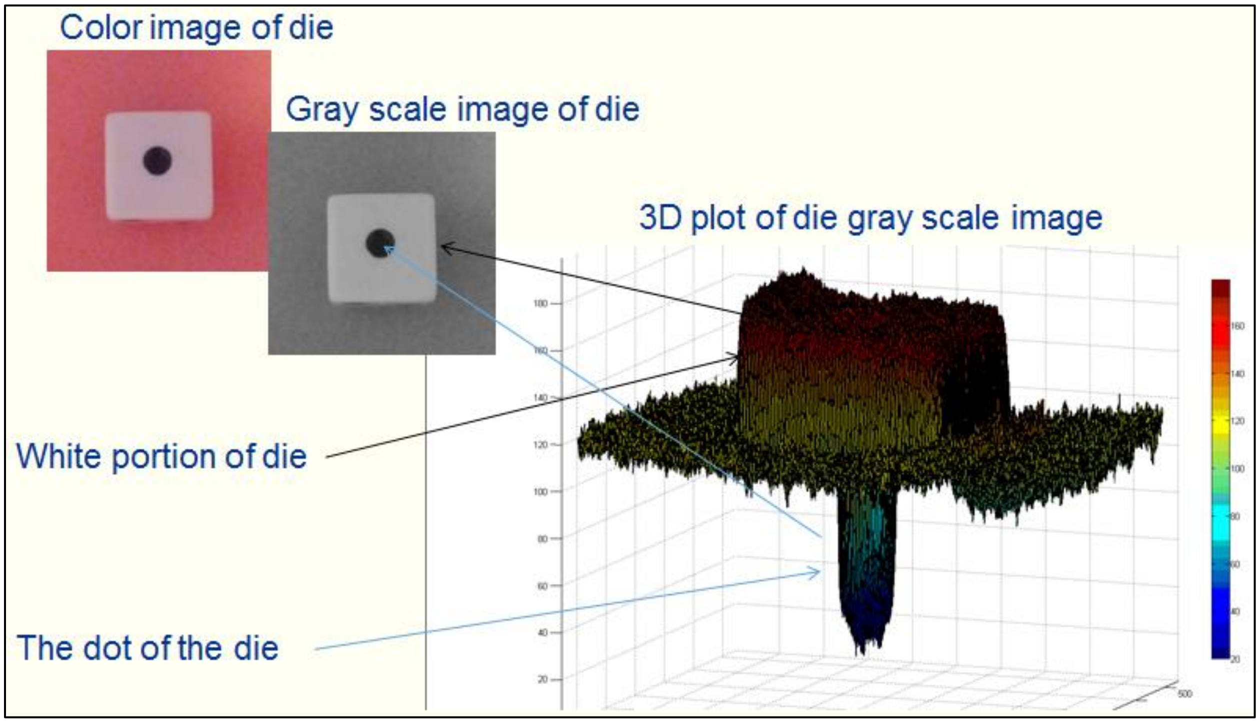


Figure 2. A Visualization of the Image Data

The Canny edge detection algorithm was chosen for this application because it has good detection and localization. Good detection means the algorithm does a good job of finding real edges, while filtering out the non-edges. Good localization means the Canny edge detector does a good job of finding the center of the edge. [2] You can read more about the Canny edge detection algorithm and its benefits on the Embedded Vision Alliance website. [3]

3.1.1 Introduction to the Canny Edge Detector

Figure 3 shows the various stages in the Canny edge detector. In the first stage, noise in the image is filtered out using a Gaussian filter, which is referred to as image smoothing or blurring. The filter removes white noise common in images captured with CMOS and CCD sensors. It removes this high frequency "popcorn noise" without significantly degrading the edges within the image.

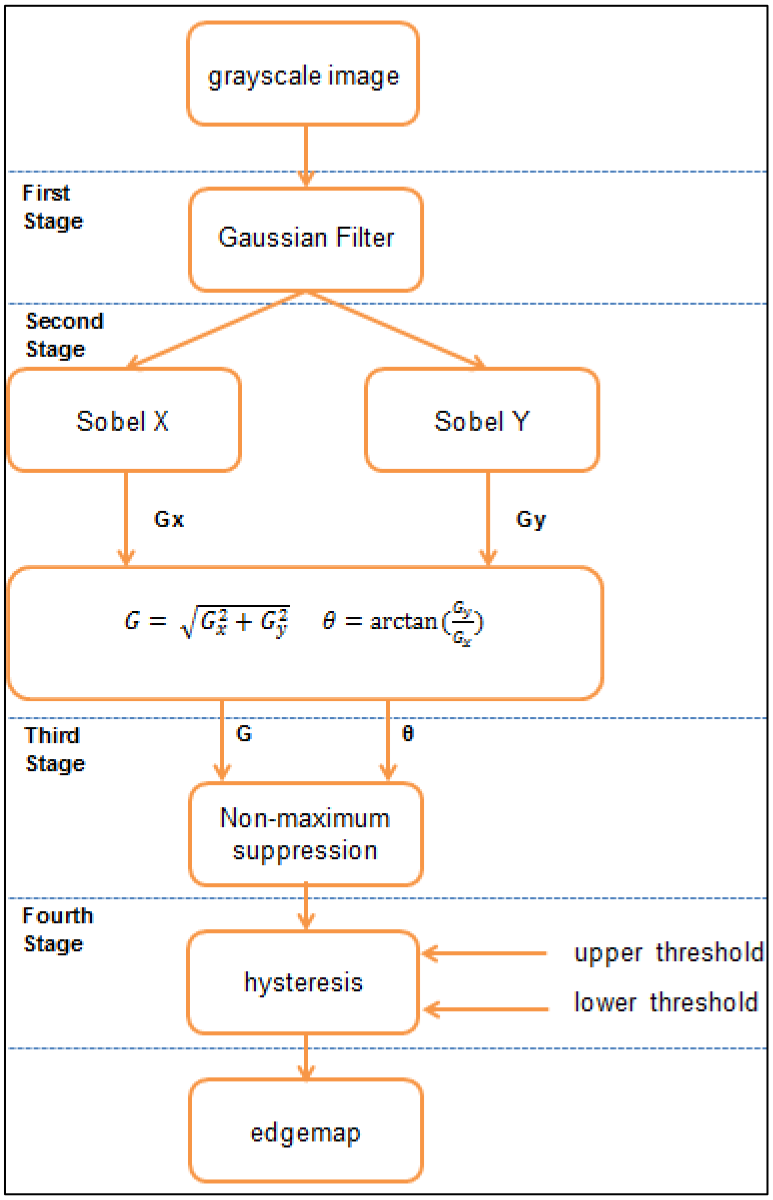


Figure 3. Block Diagram of the Canny Edge Detection Algorithm

The second stage of the Canny edge detection algorithm calculates the intensity gradient of the image using a first derivative operator. The intensity gradient of an image reflects the strength of the edges in the image. Strong edges have larger slopes and, therefore, larger intensity gradients, which is illustrated in Figure 4. The original Canny paper tested various first derivative operators; most modern Canny implementations use the Sobel operator. [4]

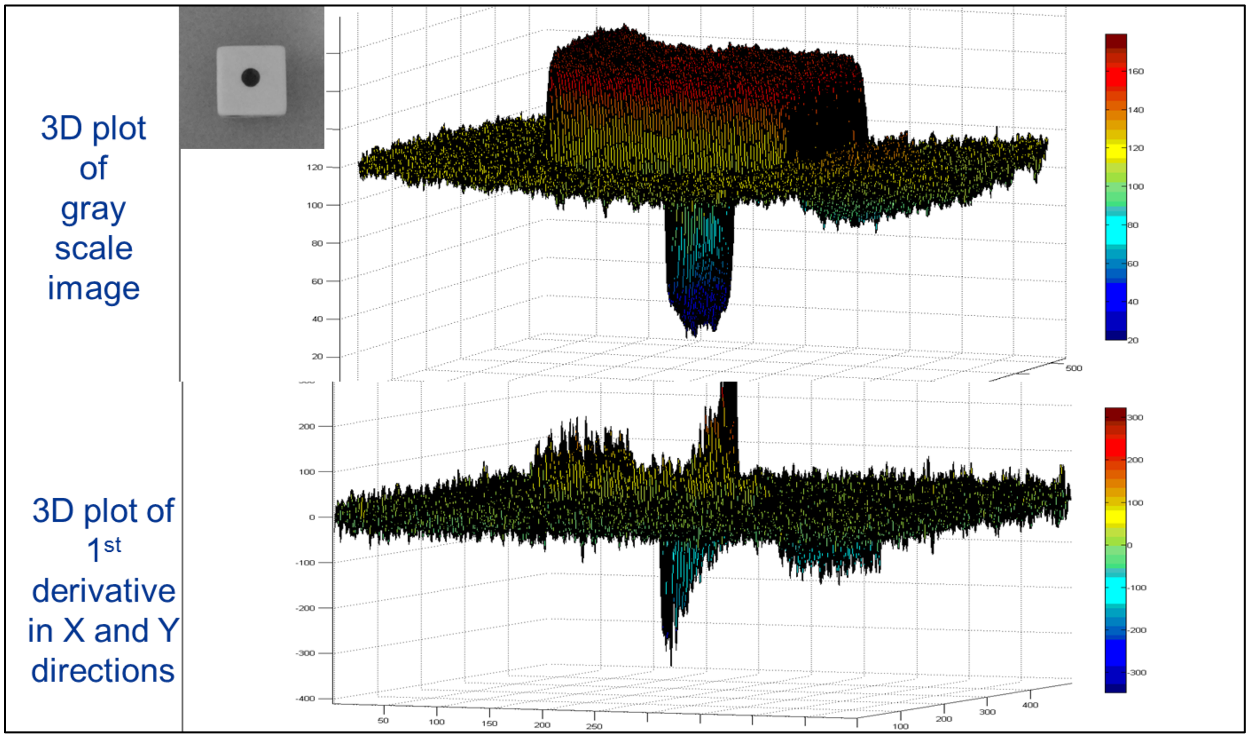


Figure 4. A Visualization of the First Derivative

The Sobel operator returns the first derivative of the image in the horizontal and vertical directions separately. This process, called convolution, is done on every pixel in the image. The result of the Sobel operator is two images, with each pixel in the first image representing the horizontal derivative of the input image and each pixel in the second image representing the vertical derivative. Sharper edges have higher magnitudes.

Convolution is illustrated in Figure 5. Convolution is the process of applying a set of weights to a group of adjacent pixels in an input image to calculate a new value for a single pixel in an output image. The weights are organized in 3×3 or 5×5 matrices (which can be larger). Each weight is multiplied by the pixel value below the weight. The multiplication products are summed into a single output value, which is then assigned to the single output pixel.

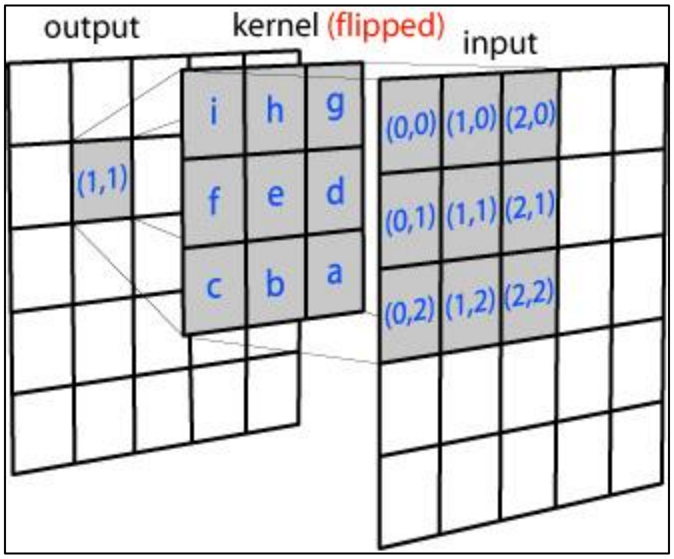


Figure 5. Convolution

(Source: http://www.songho.ca/dsp/convolution/convolution.html)

Mathematically, convolution can be described as follows:

```
Output pixel (1,1) =     input(0,0)*i + input(1,0)*h +
input(2,0)*g +
                    input(0,1)*f + input(1,1)*e + input(2,1)*d +
                    input(0,2)*c + input(1,2)*b + input(2,1)*a
```

where values "a" through "i" are the weights.

The next stage in the Canny algorithm thins the edges created by the Sobel operator using a process known as non-maximum suppression. This process removes all pixels not actually on the edge "ridge top," thereby refining the thick line into a thin line. Simply put, non-maximum suppression finds the peak of the cross section of the thick edge.

The non-maximum suppression method finds the peak of this edge by scanning across the edge using the angle data calculated by the Sobel operator. The angle data is used to scan across the line looking for a maximum magnitude. Any pixels less than the maximum magnitude are set to zero.

3.1.2 Canny Edge Detector Parameters

The final stage of the Canny edge detector is referred to as hysteresis. In this stage, the detector filters out pixels in the gradient image from the previous stage that are not part of an edge based on an upper (T2) and lower (T1) threshold, as shown in Figure 6. The algorithm first finds a gradient in the image greater than T2. It then follows the gradient, marking each pixel greater than T1 as a definite edge. The algorithm requires a gradient greater than the high threshold to start following the edge, but it will continue to follow the edge as long as the gradient stays above the low threshold.

Any pixels with gradients below T2 that are not connected to gradients above T1 are rejected as edges. This hysteresis helps to ensure that noisy edges are not broken up into multiple edge fragments. [2]

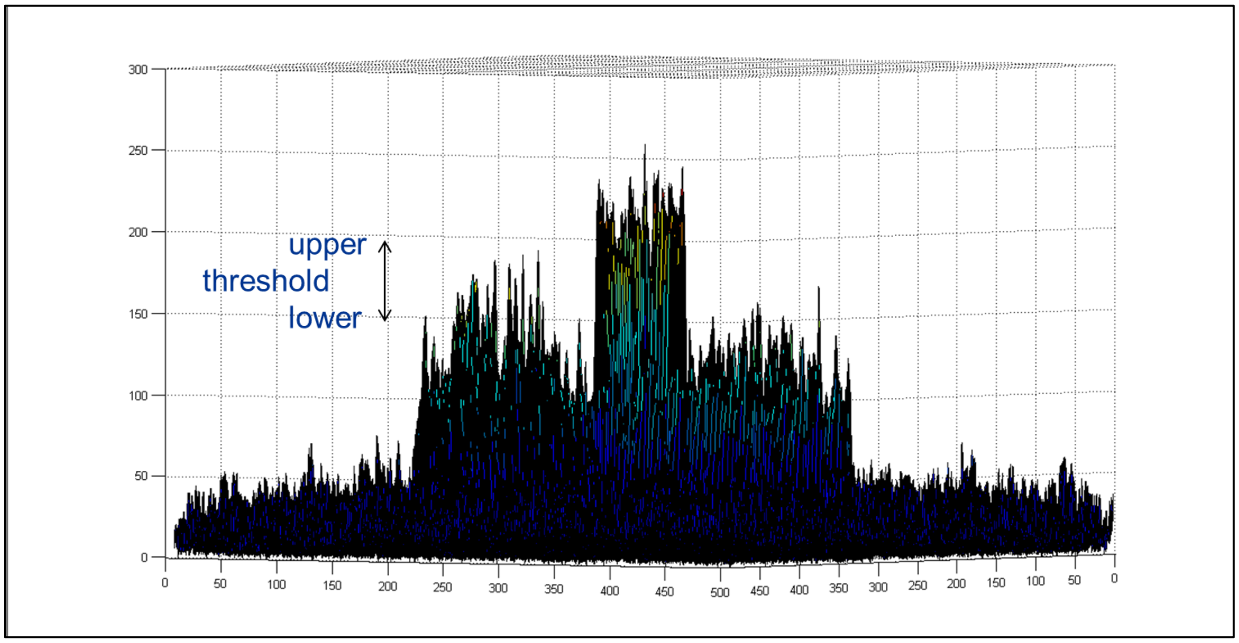


Figure 6. Where to set the Canny Thresholds

3.2 Grouping Region Boundaries to Contours

The Canny edge detector marks each pixel as an edge or non-edge. At this point each pixel is still an individual entity. In the next layer of the design, pixels marked as edges surrounding the same region are grouped together as contours.

3.2.1 What is a Contour?

The dictionary definition of *contour* is "The outline of a figure, body, or mass." The definition of a *contour line* is "A line joining points of equal elevation on a surface." The computer vision definition of an edge is a "difference in gray scale value between two pixels." Combining these definitions, in computer vision a contour is a line joining points of equal difference between gray scale pixel values in an image, outlining a figure, body, or mass in that image.

Contours are chains of connected edge pixels that surround a region in an image. In computer vision, a contour has a length, an area (called a region), and a shape. [4]

3.2.2 Grouping Edges into Contours

Contours are formed by grouping neighboring edge pixels, which are combined to form a contour by starting with one edge pixel and looking for others next to the first edge pixel. *Connectivity* rules determine how many candidates are examined when looking for neighboring edge pixels. "There are two common ways of defining connectivity: 4-way (or 4-neighborhood) where only immediate neighbors are analyzed for connectivity; or 8-way (or 8-neighborhood) where all the eight pixels surrounding a chosen pixel are analyzed for connectivity." [1] The ADI contour library used for this demo uses 8-way connectivity, which is illustrated in Figure 7.

Contours are described via "chain codes" that encode the position of each successive edge pixel relative to the previous one. The chain code is comprised of a sequence of numbers that relate one edge pixel to the next in terms of the direction of travel between them, similar to how compass directions are used to describe a route on a map. For example, the chain code for the contour of a square using 8-way connectivity would be 2-2-2-4-4-4-6-6-6-0-0-0.

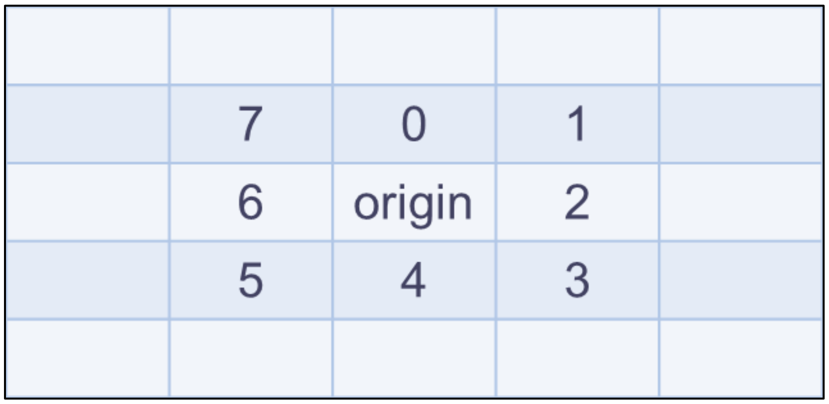


Figure 7. 8-way Connectivity

3.2.3 Finding the Area of a Contour

Using chain codes to describe contours makes it easy to calculate properties of the contour. For example, finding the length of the contour is as simple as counting the nodes in the chain code. Finding the area of the region bounded by the contour is accomplished by counting pixels within the contour. Starting at the pixel with the least Y value and incrementing through Y values, we find the minimum X value and maximum X value for each Y value. X maximum – X minimum + 1 yields the number of pixels in the contour for that Y value. The number of pixels is accumulated throughout the Y values in the contour.

3.3 Classifying Contours into Dice Dots

To find dots, the contours are evaluated based on their area and boundary size using a three-stage filter cascade, shown in Figure 8. A filter cascade is simply a series of filters where the output of one filter is fed into the input of the next. Each filter discards certain contours based on their properties.

The first filter in the series discards any contours that have an area too big or too small. The next discards any contours that do not fit the mathematical model of the area of a circle inscribed in a square. The final filter discards any contours that are not symmetrical. The following section describes each filter in more detail.

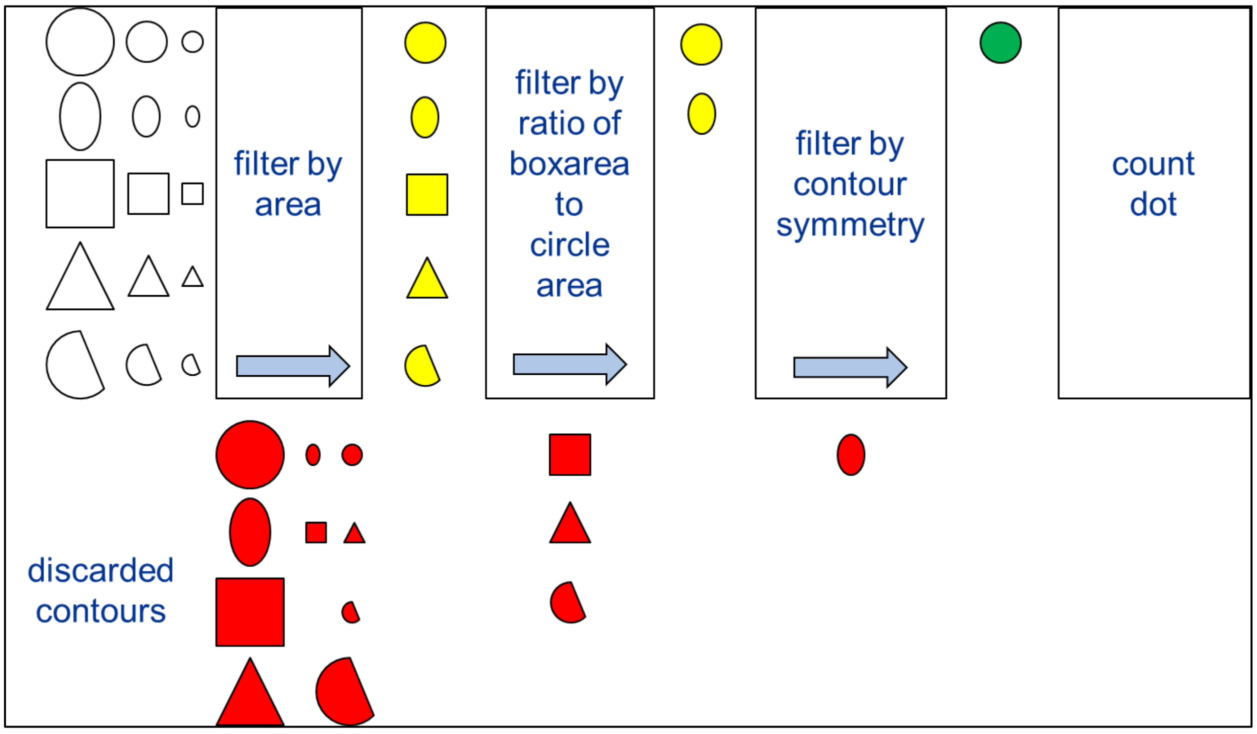

filter by area
filter by ratio of boxarea to circle area
filter by contour symmetry
count dot
discarded contours

Figure 8. Block Diagram of the Dot Classifier

3.3.1 Filtering based on Area

The first filter discards contours based on area. Contours too large or too small are discarded, while those within the specified range are passed to the next filter in the cascade. The size of the dot varies depending on how far the die is from the camera. As the camera gets closer to the die, the dot image—and therefore its contour—get bigger. The area range therefore affects the distance at which dice dots can be found.

3.3.2 The Formula for a Circle Inscribed in a Square

Various mathematical models for circles were evaluated while developing this design. Some models required too much precision while others were too compute-intensive. For example, using the ratio of circle area to circumference was evaluated, but it requires sub-pixel area and circumference measurements to be accurate. The Hough transform was also evaluated, but it requires far too many CPU cycles to allow real-time operation at the desired frame rate.

The inscribed circle model, shown in Figure 9, was chosen because it does not require sub-pixel precision or floating-point math, and so consumes the least amount of CPU cycles for the accuracy of the model. The box area is measured by simply subtracting the minimum X and Y values in the contour from the maximum X and Y values. This measurement results in the width and height of the rectangle surrounding the contour (referred to as a bounding rectangle). The area of the rectangle is calculated by multiplying the height times the width.

The area of the bounding rectangle is used to calculate the area of a circle inscribed in the rectangle using the mathematical model. This calculated area is then divided by the measure area of the contour. The resulting ratio represents how close the contour matches the model of a circle. A perfect match results in a ratio of 1.0. The calculation is done using integers, and a ratio of 100 corresponds to a perfect match. Anything greater or less than 100 (1.0) does not match the model to some degree. The filter discards contours with ratios far from 100 (1.0) and passes contours close to 100 (1.0). The minimum and maximum ratios used by the filter are hard-coded constants.

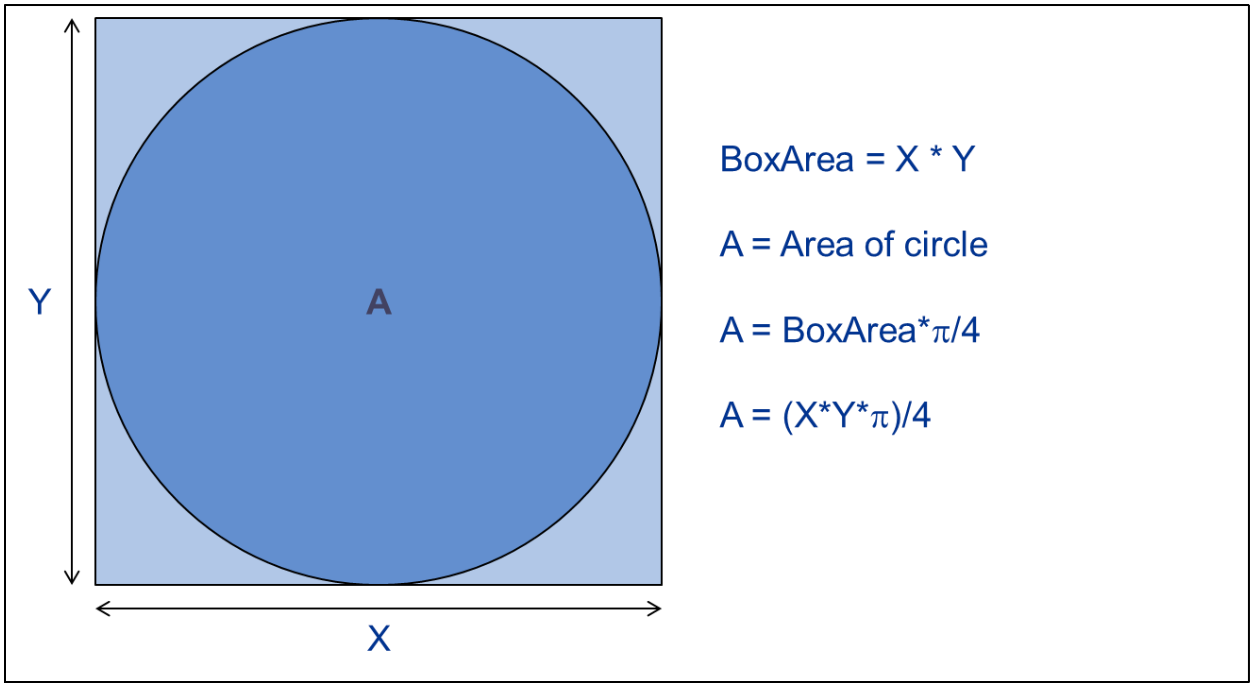


Figure 9. Formula for a Circle Inscribed in a Square

3.3.3 Checking for Symmetry

The final filter checks for contour symmetry. The formula for a circle inscribed in a square does not apply to an oval inscribed in a rectangle. The formula only works if the contour bounding rectangle has sides of equal length (a square). The symmetry filter discards contours that do not have symmetrical bounding rectangles (the ratio of bounding rectangle width over height is equal to 1.0). The ratio is calculated using integer arithmetic to avoid unnecessary floating-point operations. A ratio of 10 corresponds to 1.0. The filter discards contours with ratios far from 10 (1.0) and passes contours close to 10 (1.0). The minimum and maximum ratios used by the filter are hard-coded constants.

4. Implementing the Algorithm on the BF609

The Analog Devices BF609 contains two 500 MHz Blackfin DSP/CPU cores, a computer vision accelerator called the Pipelined Vision Processor (PVP) and various peripherals as shown in Figure 10. This discussion will concentrate on the Blackfin cores and the PVP. Additional information on the BF609 is available [5].

The PVP hardware is complemented by a library called the "Blackfin Vision Analytics Toolbox" or VAT [6]. "VAT provides building blocks for vision based automated applications." [6] The VAT library provides functions to post process the data stream coming from the PVP.

PVP hardware configuration and management is accomplished using the PVP device driver. The device driver presents an API referred to in the rest of this document as the PVP API.

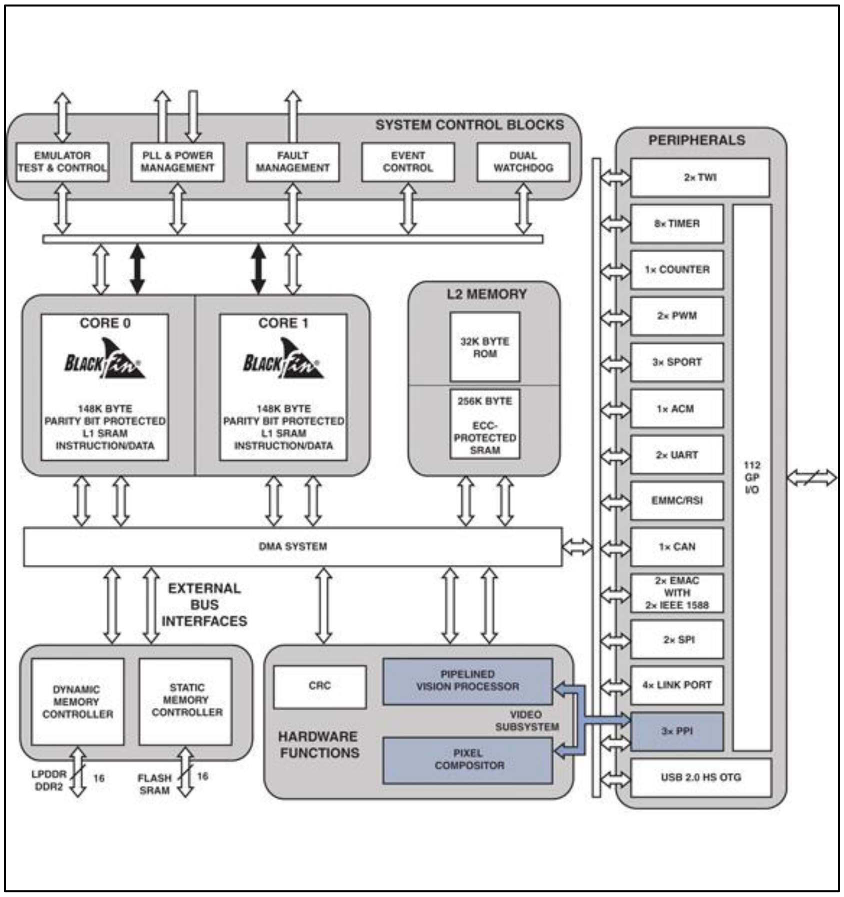


Figure 10. A Block Diagram of the Analog Devices BF609

4.1 Using the PVP and ADI Edge Trace Library to Detect Edges

4.1.1 Introduction to the PVP

A diagram of the Pipelined Vision Processor illustrating all its processing blocks is shown in Figure 11. "The Pipelined Vision Processor (PVP) provides a set of 12 high-performance signal processing blocks that can be flexibly combined to form streaming data processing pipes. These blocks are optimized for tasks typical of video and image processing, analytics (e.g., advanced driver assistance systems), robotics, and 2-dimensional vector applications. The PVP works in conjunction with the processor core(s). It is optimized for convolution and wavelet-based object detection, classification, tracking, and verification algorithms. The PVP bundles a set of processing blocks required for high-speed 2-dimensional digital signal processing." [7]

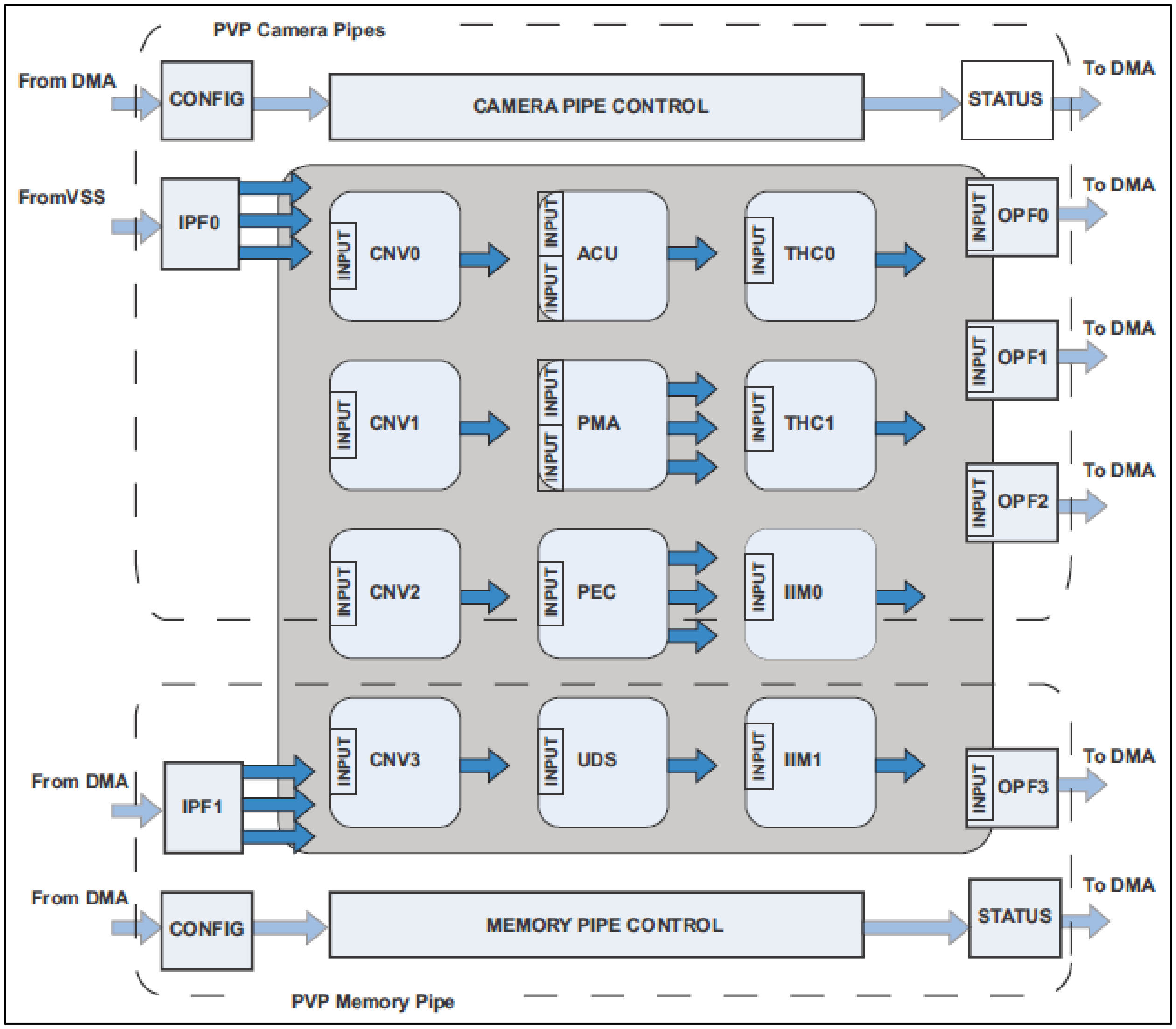


Figure 11. Block Diagram of the Pipeline Vision Processor

4.1.2 PVP Configuration

Figure 12 shows the PVP configured to perform Canny edge detection. Five of the 12 available blocks in the PVP are used. (The blocks used are shown in green.) The gray boxes with numbers in them show which ports are used to connect the blocks. The IPF0 and OPF0 blocks are DMA blocks used to stream data into and out of the PVP.

4.1.3 PVP Edge Detection Functionality

Convolution block CNV2 is configured as a Gaussian filter to reduce image noise. Convolution blocks 0 and 1 are configured with Sobel X and Y kernels. The X and Y data is converted in the Polar Magnitude and Angle (PMA) block into polar form (magnitude and angle) and fed into the Pixel Edge Classifier (PEC) block. The PEC block feeds the magnitude and angle information into a first derivative operator to find edges, then performs edge thinning using non-maximum suppression, and finally encodes relative edge strengths for a post-processing operation to eliminate streaking (see Section 4.1.3.3, [Page 16]) using hysteresis. The post-processing operation is performed by code running on the CPU core.

The following sections describe the BF609 algorithm implementation in more detail.

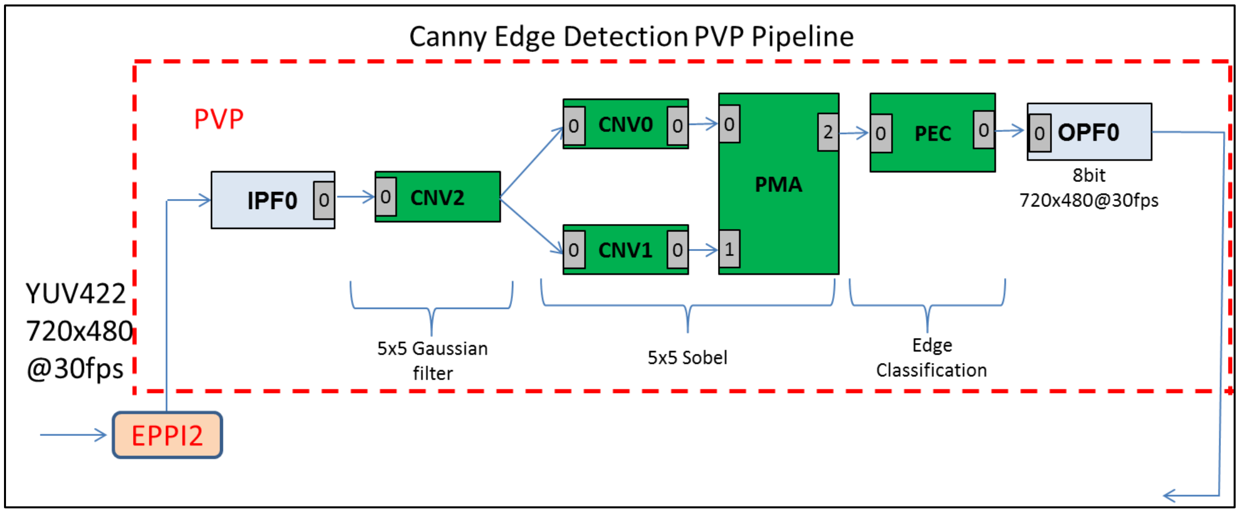
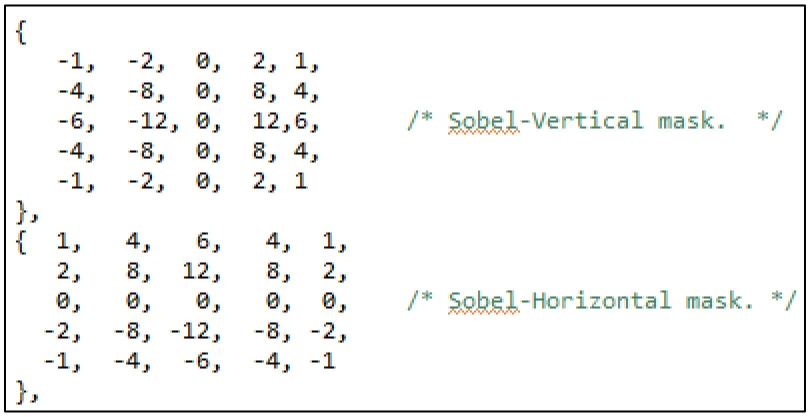


Figure 12. Canny Edge Detection PVP Configuration

4.1.3.1 Convolution Blocks for Sobel (CNV0, CNV1)

Convolution blocks CNV0 and CNV1 are configured as a Sobel differential gradient operator. Configuration is done by loading the convolution blocks with Sobel horizontal and vertical kernels. The Sobel kernels (or "masks") are shown in Figure 13. The horizontal kernel performs the first derivative operation on the image in the horizontal direction. The vertical kernel performs the first derivative operation on the image in the vertical direction. [8]



```
{
   -1,  -2,  0,  2, 1,
   -4,  -8,  0,  8, 4,
   -6,  -12, 0,  12,6,      /* Sobel-Vertical mask.  */
   -4,  -8,  0,  8, 4,
   -1,  -2,  0,  2, 1
},
{  1,   4,   6,   4,  1,
   2,   8,  12,   8,  2,
   0,   0,   0,   0,  0,     /* Sobel-Horizontal mask. */
  -2,  -8, -12,  -8, -2,
  -1,  -4,  -6,  -4, -1
},
```

Figure 13. Sobel Kernels or Masks

4.1.3.2 Polar Magnitude and Angle Block for Cartesian to Polar (PMA)

The Polar Magnitude and Angle Block takes in the Sobel horizontal and vertical outputs and combines them into a polar form (magnitude and angle) using the equations in Figure 14. [7]

$$Magnitude = \sqrt{|x|^2 + |y|^2} \qquad \varphi = \arctan(\frac{y}{x})$$

Figure 14. Polar Magnitude and Angle Block Equations

4.1.3.3 Pixel Edge Classification for First Derivative and Non-Maximum Suppression

The Pixel Edge Classifier (PEC) is set up in mode 1. The PEC takes in polar data from the PMA representing rough edges and performs edge thinning using non-maximum suppression. "The Canny algorithm uses hysteresis thresholding. When a single threshold value is used, the fluctuations of the gradient magnitude (due to noise) above and below the threshold value results in pixels being classified as edges and non-edges. In such cases the edge line appears broken. This phenomenon is commonly referred to as streaking. PEC-1 eliminates streaking by comparing the magnitude against two threshold registers. If the magnitude lies below (TL), then the pixel is called *no edge*. If the magnitude lies above (TH) then the pixel is termed as *strong edge*. If it lies between TL and TH, then the pixel is termed as *weak edge*. The post-processing software can connect the weak edge to strong edge to create an unbroken edge line." [7]

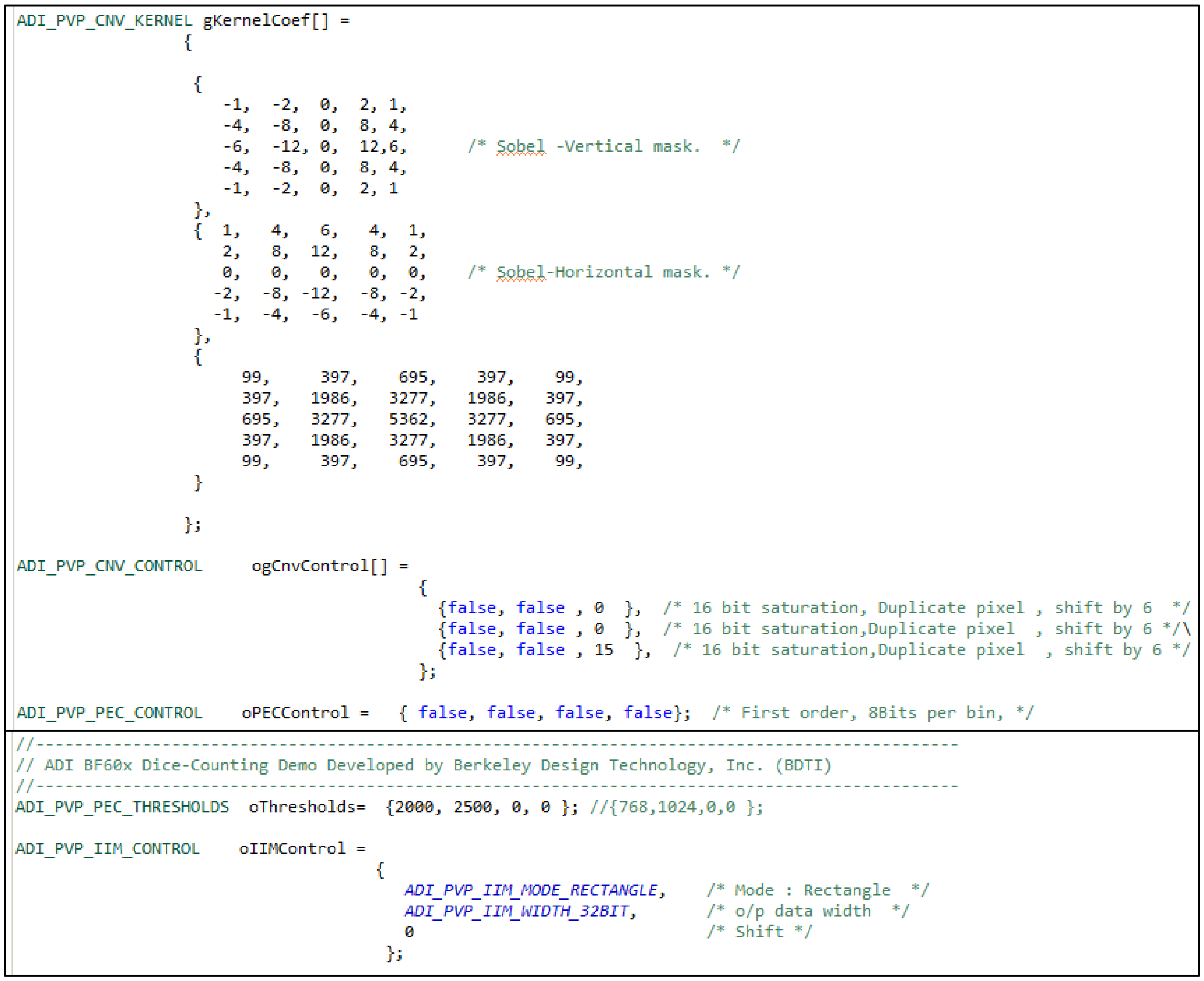
4.1.4 Blackfin Vision Analytics Toolbox (VAT)—Hysteresis Thresholding

The output of the PEC classifies each pixel as either *no edge, weak edge,* or *strong edge*. The EdgeDetect function in the VAT library [6] uses these pixel classifications to perform hysteresis thresholding. For a *weak edge* to be reclassified as a *strong edge,* it must be connected to a *strong edge*. *Weak edges* not connected *to strong edges* are reclassified as *no edges*.

The code works by scanning all the edges and looking for a *strong edge*. It then looks for *weak edges* connected to the *strong edge* using the 8-way connectivity rules described earlier. Any *weak edges* connected to the *strong edge* are upgraded to *strong edges*. The process then repeats. Any *weak edges* left over are reclassified as *no edges*. [7]

4.1.5 Setting Canny Parameters in the PVP

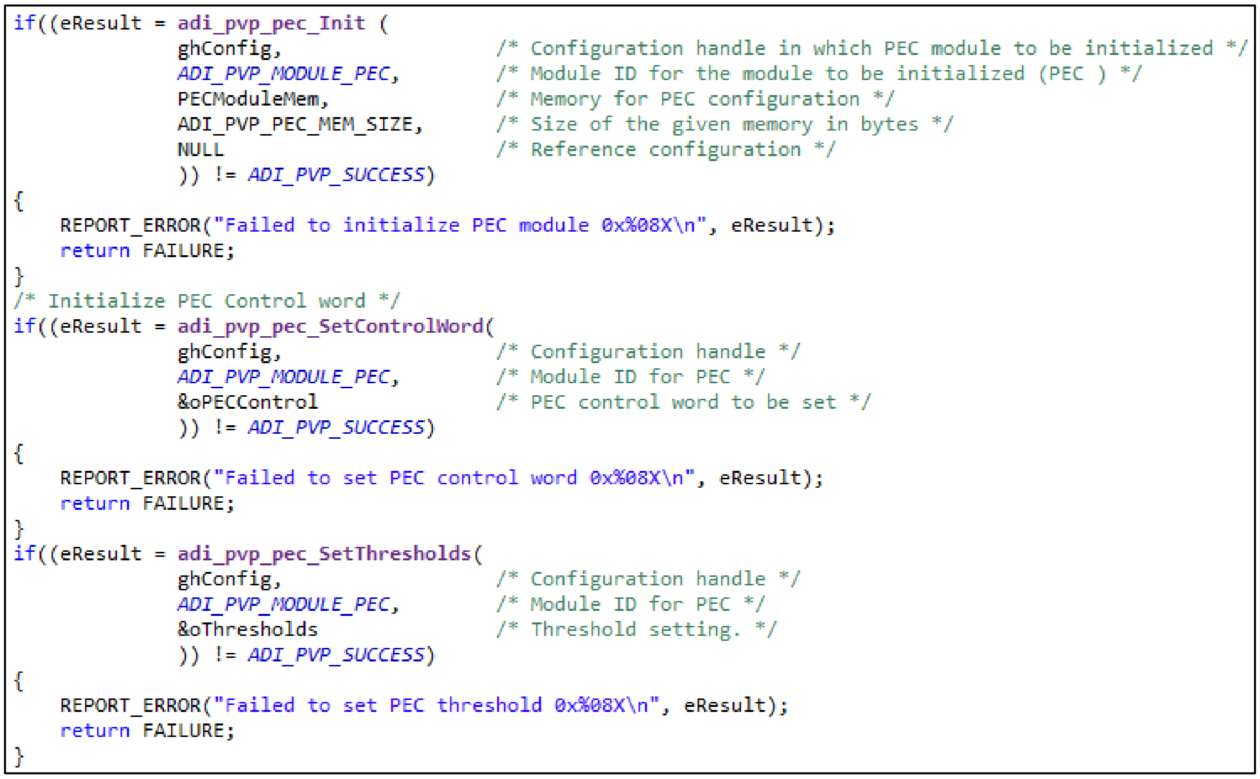
The PVP is configured using registers. Each block has its own register set defining specific parameters. The PVP API and Edge Detection example make it easy to set PVP parameters by putting all the parameters in one file called `PVPInput.c`. A portion of `PVPInput.c` is shown in Figure 15.



```
ADI_PVP_CNV_KERNEL gKernelCoef[] =
                 {

                  {
                     -1,  -2,  0,  2, 1,
                     -4,  -8,  0,  8, 4,
                     -6,  -12, 0,  12,6,       /* Sobel -Vertical mask.  */
                     -4,  -8,  0,  8, 4,
                     -1,  -2,  0,  2, 1
                  },
                  {  1,   4,   6,   4,  1,
                     2,   8,  12,   8,  2,
                     0,   0,   0,   0,  0,     /* Sobel-Horizontal mask. */
                    -2,  -8, -12,  -8, -2,
                    -1,  -4,  -6,  -4, -1
                  },
                  {
                       99,     397,    695,    397,    99,
                       397,   1986,   3277,   1986,   397,
                       695,   3277,   5362,   3277,   695,
                       397,   1986,   3277,   1986,   397,
                       99,     397,    695,    397,    99,
                  }

                 };

ADI_PVP_CNV_CONTROL     ogCnvControl[] =
                                        {
                                          {false, false , 0  },  /* 16 bit saturation, Duplicate pixel , shift by 6  */
                                          {false, false , 0  },  /* 16 bit saturation,Duplicate pixel  , shift by 6 */\
                                          {false, false , 15  },  /* 16 bit saturation,Duplicate pixel  , shift by 6 */
                                        };

ADI_PVP_PEC_CONTROL    oPECControl =   { false, false, false, false};  /* First order, 8Bits per bin, */
//-------------------------------------------------------------------------------------
// ADI BF60x Dice-Counting Demo Developed by Berkeley Design Technology, Inc. (BDTI)
//-------------------------------------------------------------------------------------
ADI_PVP_PEC_THRESHOLDS  oThresholds=  {2000, 2500, 0, 0 }; //{768,1024,0,0 };

ADI_PVP_IIM_CONTROL    oIIMControl =
                                    {
                                      ADI_PVP_IIM_MODE_RECTANGLE,    /* Mode : Rectangle  */
                                      ADI_PVP_IIM_WIDTH_32BIT,       /* o/p data width  */
                                      0                              /* Shift */
                                     };
```

Figure 15. Configuring PVP Block Parameters in PVPInput.c

The ADI Canny Edge Detection Example performs the actual initialization of the PVP in the module `PVPInit.c` using the parameters in `PVPInput.c`. Figure 16 shows how to use the PVP API functions to initialize the PVP PEC block. Note the *oThresholds* array being used by the `adi_pvp_pec_SetThresholds()` function call.



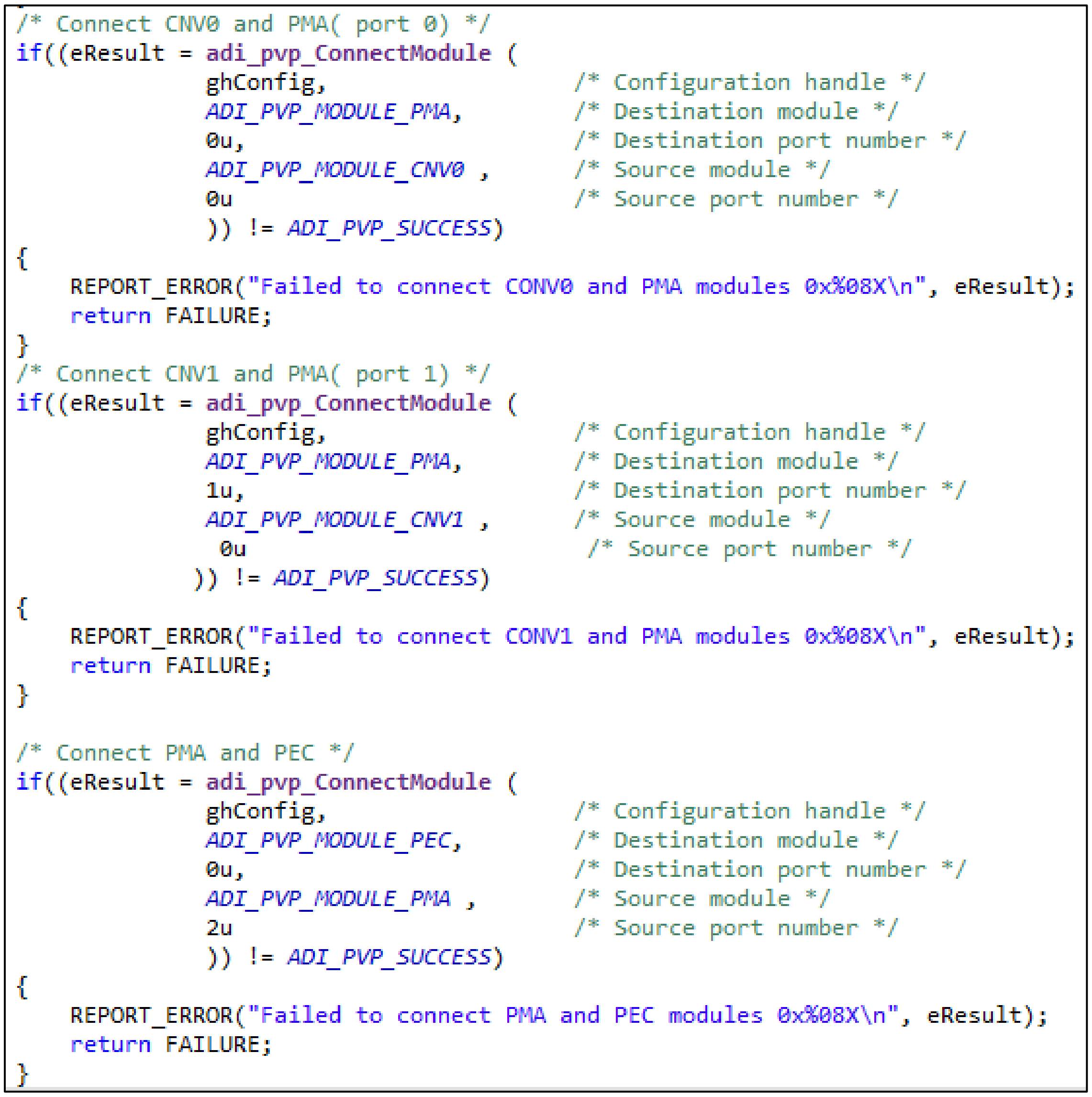
```
if((eResult = adi_pvp_pec_Init (
              ghConfig,                  /* Configuration handle in which PEC module to be initialized */
              ADI_PVP_MODULE_PEC,        /* Module ID for the module to be initialized (PEC ) */
              PECModuleMem,              /* Memory for PEC configuration */
              ADI_PVP_PEC_MEM_SIZE,      /* Size of the given memory in bytes */
              NULL                       /* Reference configuration */
              )) != ADI_PVP_SUCCESS)
{
    REPORT_ERROR("Failed to initialize PEC module 0x%08X\n", eResult);
    return FAILURE;
}
/* Initialize PEC Control word */
if((eResult = adi_pvp_pec_SetControlWord(
              ghConfig,                  /* Configuration handle */
              ADI_PVP_MODULE_PEC,        /* Module ID for PEC */
              &oPECControl               /* PEC control word to be set */
              )) != ADI_PVP_SUCCESS)
{
    REPORT_ERROR("Failed to set PEC control word 0x%08X\n", eResult);
    return FAILURE;
}
if((eResult = adi_pvp_pec_SetThresholds(
              ghConfig,                  /* Configuration handle */
              ADI_PVP_MODULE_PEC,        /* Module ID for PEC */
              &oThresholds               /* Threshold setting. */
              )) != ADI_PVP_SUCCESS)
{
    REPORT_ERROR("Failed to set PEC threshold 0x%08X\n", eResult);
    return FAILURE;
}
```

Figure 16. Initializing the PEC using the PVP API in PVPInit.c

4.1.6 Configuring the PVP

`PVPInit.c` initializes and configures the PVP. Configuring the PVP involves connecting the PVP modules together using the PVP API. Each module has various input and output ports, and each port is capable of being connected to various other modules.

Figure 17 shows a code snippet from `PVPInit.c`, which shows the configuration for connecting CNV0 and CNV1 into the PMA, and the PMA into the PEC. The PVP API call `adi_pvp_ConnectModule()` is used to connect a source module and port to a destination and port. This process is repeated for all the required PVP blocks. The PVP contains *input* and *output* blocks used to perform DMA transfers to and from memory or other hardware blocks within the chip. The input blocks are labeled IPF and the output blocks are labeled OPF. The PVP API treats these just like any other block. Connecting to either an IPF or OPF block is done using the `adi_pvp_ConnectModule()` function.



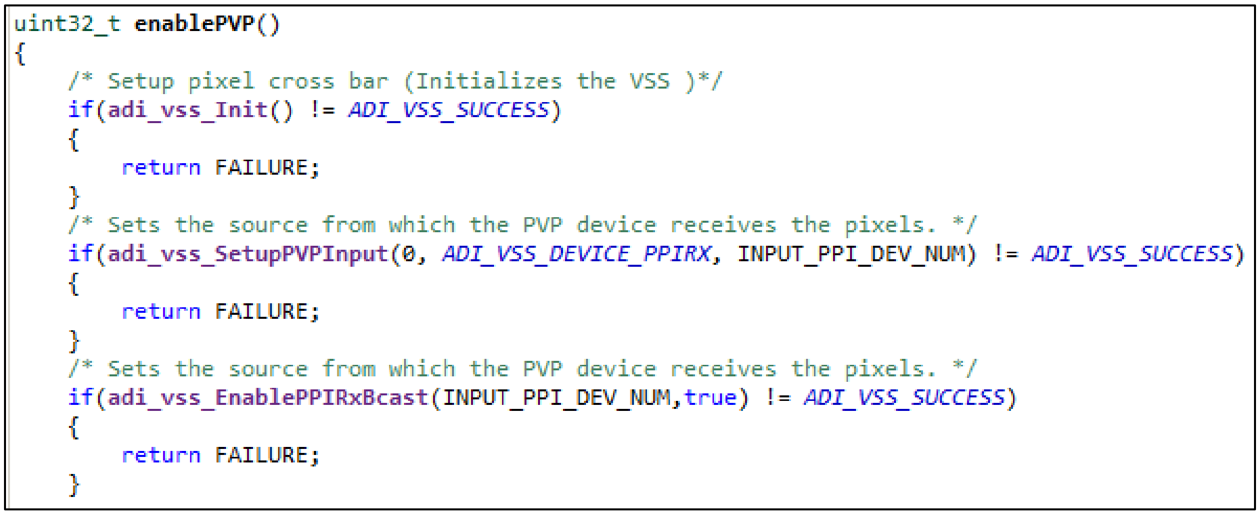
```
/* Connect CNV0 and PMA( port 0) */
if((eResult = adi_pvp_ConnectModule (
              ghConfig,               /* Configuration handle */
              ADI_PVP_MODULE_PMA,     /* Destination module */
              0u,                     /* Destination port number */
              ADI_PVP_MODULE_CNV0 ,   /* Source module */
              0u                      /* Source port number */
              )) != ADI_PVP_SUCCESS)
{
    REPORT_ERROR("Failed to connect CONV0 and PMA modules 0x%08X\n", eResult);
    return FAILURE;
}
/* Connect CNV1 and PMA( port 1) */
if((eResult = adi_pvp_ConnectModule (
              ghConfig,               /* Configuration handle */
              ADI_PVP_MODULE_PMA,     /* Destination module */
              1u,                     /* Destination port number */
              ADI_PVP_MODULE_CNV1 ,   /* Source module */
               0u                      /* Source port number */
             )) != ADI_PVP_SUCCESS)
{
    REPORT_ERROR("Failed to connect CONV1 and PMA modules 0x%08X\n", eResult);
    return FAILURE;
}

/* Connect PMA and PEC */
if((eResult = adi_pvp_ConnectModule (
              ghConfig,               /* Configuration handle */
              ADI_PVP_MODULE_PEC,     /* Destination module */
              0u,                     /* Destination port number */
              ADI_PVP_MODULE_PMA ,    /* Source module */
              2u                      /* Source port number */
              )) != ADI_PVP_SUCCESS)
{
    REPORT_ERROR("Failed to connect PMA and PEC modules 0x%08X\n", eResult);
    return FAILURE;
}
```

Figure 17. Using the PVP API to configure the PVP

4.1.7 Getting Frames Into the PVP

"The enhanced parallel peripheral interface (EPPI) is a half-duplex, bidirectional port with a dedicated clock pin and three frame sync (FS) pins. It can support direct connections to active TFT LCD, parallel A/D and D/A converters, video encoders and decoders, image sensor modules and other general-purpose peripherals." [7] The BF609 uses one of its EPPIs to interface with the CMOS sensor. The particular EPPI depends on the hardware. The FinBoard connects the sensor to EPPI2 and the output video encoder to EPPI0. The ADSP-BF609 EZ-KIT Lite connects the sensor to EPPI0 and the output video encoder to EPPI2. This information can be found in `EdgeDetection.h`.

Figure 18 is a code snippet from PVPInit showing the PVP input source configuration. The PVP API call `adi_vss_SetupPVPInput()` connects the PVP input to the EPPI port specified in the constant `INPUT_PPI_DEV_NUM` (set in `EdgeDetection.h`). The `adi_vss_EnablePPIRxBcast()` call enables the PPI RX broadcast bit. This will cause the EPPI to broadcast an interrupt when a video frame is received from the sensor.

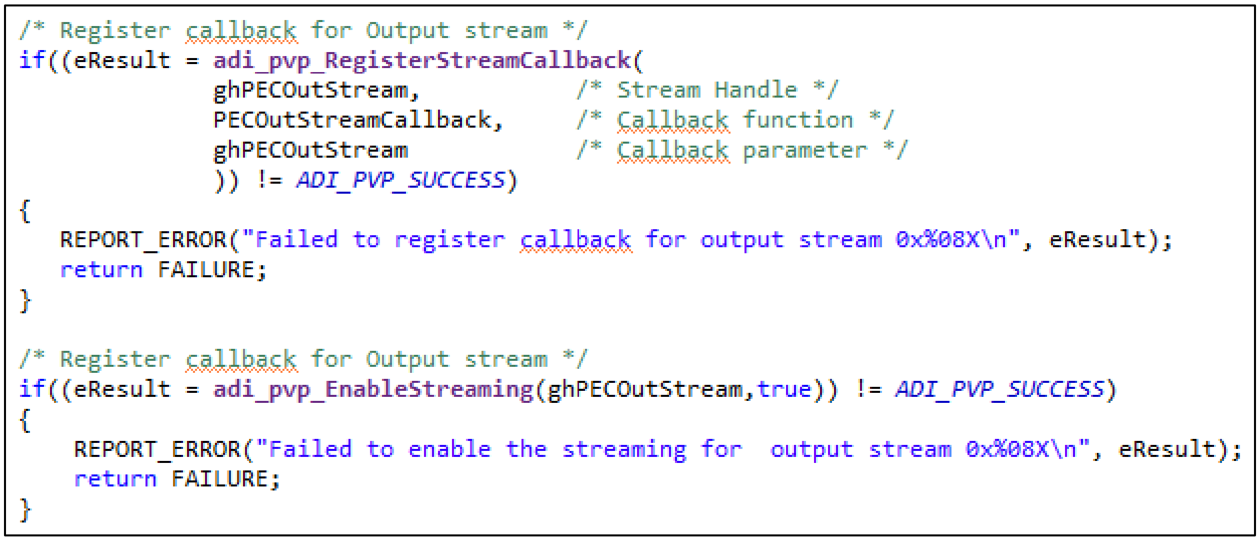


```
uint32_t enablePVP()
{
    /* Setup pixel cross bar (Initializes the VSS )*/
    if(adi_vss_Init() != ADI_VSS_SUCCESS)
    {
        return FAILURE;
    }
    /* Sets the source from which the PVP device receives the pixels. */
    if(adi_vss_SetupPVPInput(0, ADI_VSS_DEVICE_PPIRX, INPUT_PPI_DEV_NUM) != ADI_VSS_SUCCESS)
    {
        return FAILURE;
    }
    /* Sets the source from which the PVP device receives the pixels. */
    if(adi_vss_EnablePPIRxBcast(INPUT_PPI_DEV_NUM,true) != ADI_VSS_SUCCESS)
    {
        return FAILURE;
    }
```

Figure 18. Initializing the PVP Input Source

4.1.8 Getting Frames From the PVP

Data exits the PVP via a DMA transfer from the PVP to memory. An example of this can be found in `PVPInit.c` within the function `PECOutStreamCallback()`. Figure 19 shows how the callback function is registered with the PVP DMA controller using the PVP API. Once the callback function is registered and the stream is enabled using the `adi_pvp_EnableStreaming()` call, the PVP DMA controller's interrupt handler will start calling the callback function and passing it frames of data. The function `PECOutStreamCallback()` in `PVPInit.c` accepts the buffer, then submits a new clean buffer from the buffer pool using the `adi_pvp_SubmitBuffer()` function.



```
/* Register callback for Output stream */
if((eResult = adi_pvp_RegisterStreamCallback(
              ghPECOutStream,           /* Stream Handle */
              PECOutStreamCallback,     /* Callback function */
              ghPECOutStream            /* Callback parameter */
              )) != ADI_PVP_SUCCESS)
{
   REPORT_ERROR("Failed to register callback for output stream 0x%08X\n", eResult);
   return FAILURE;
}

/* Register callback for Output stream */
if((eResult = adi_pvp_EnableStreaming(ghPECOutStream,true)) != ADI_PVP_SUCCESS)
{
    REPORT_ERROR("Failed to enable the streaming for  output stream 0x%08X\n", eResult);
    return FAILURE;
}
```

Figure 19. Registering a Callback Function to Receive Data from the PVP (in PVPInit.c)

4.2 Overview of ADI Canny Edge Detection Example

The BDTI Dice Dot-Counting Demo and Reference Design is built on top of the ADI Canny Edge Detection Example shown in Figure 20. Edge detection is a very common segmentation layer for many computer vision applications. A whole category of computer vision applications is based on detecting the boundary of a region and fitting the boundary to a mathematical model (this is the category used for the dice counting demo). [1]

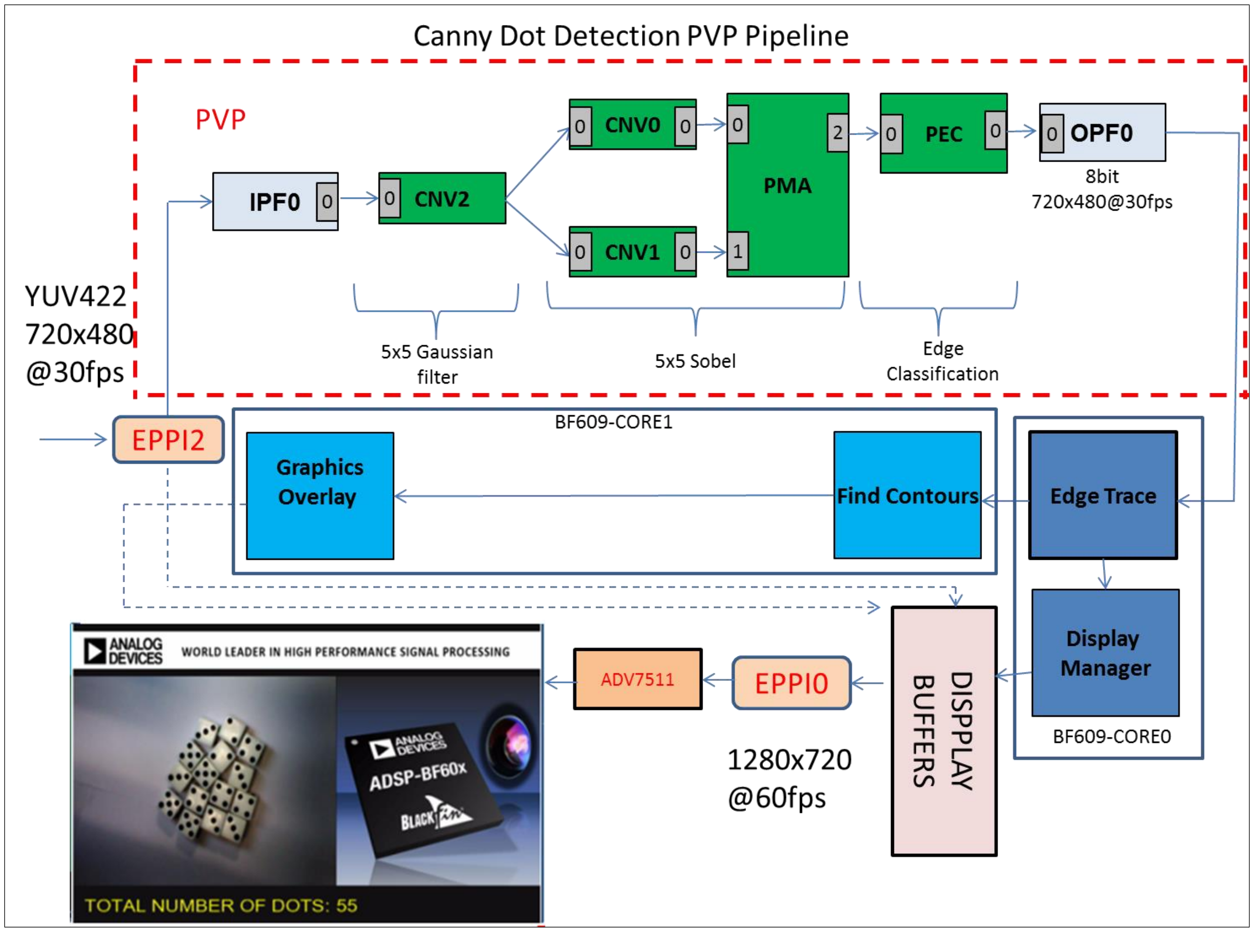


Figure 20. Block Diagram of the ADI Canny Edge Detection Example

4.2.1 Distribution Across Cores

The ADI Canny Edge Detection Example is a multi-core application with two entry points. Figure 21 shows how the application is distributed between the cores. Core 0 takes in video frames from the PVP and transforms them to edge maps by calling the `EdgeDetect` function in the VAT library. The edge map buffers are handed over to Core 1 via an ADI multi-core library. The multi-core library manages all the signaling and buffer exchanges between cores.

When Core 0 has an edge map buffer ready to send to Core 1, it sends a `DSP_GFXCONTOUR` command to Core 1. The `dsp_command_loop()` function in Core 1 receives the `DSP_GFXCONTOUR` command along with the edge map buffer and passes the buffer to the ADI find contours library. The find contours library generates data for the ADI OpenGL graphics library, which displays a graphics overlay on top of the live video feed.

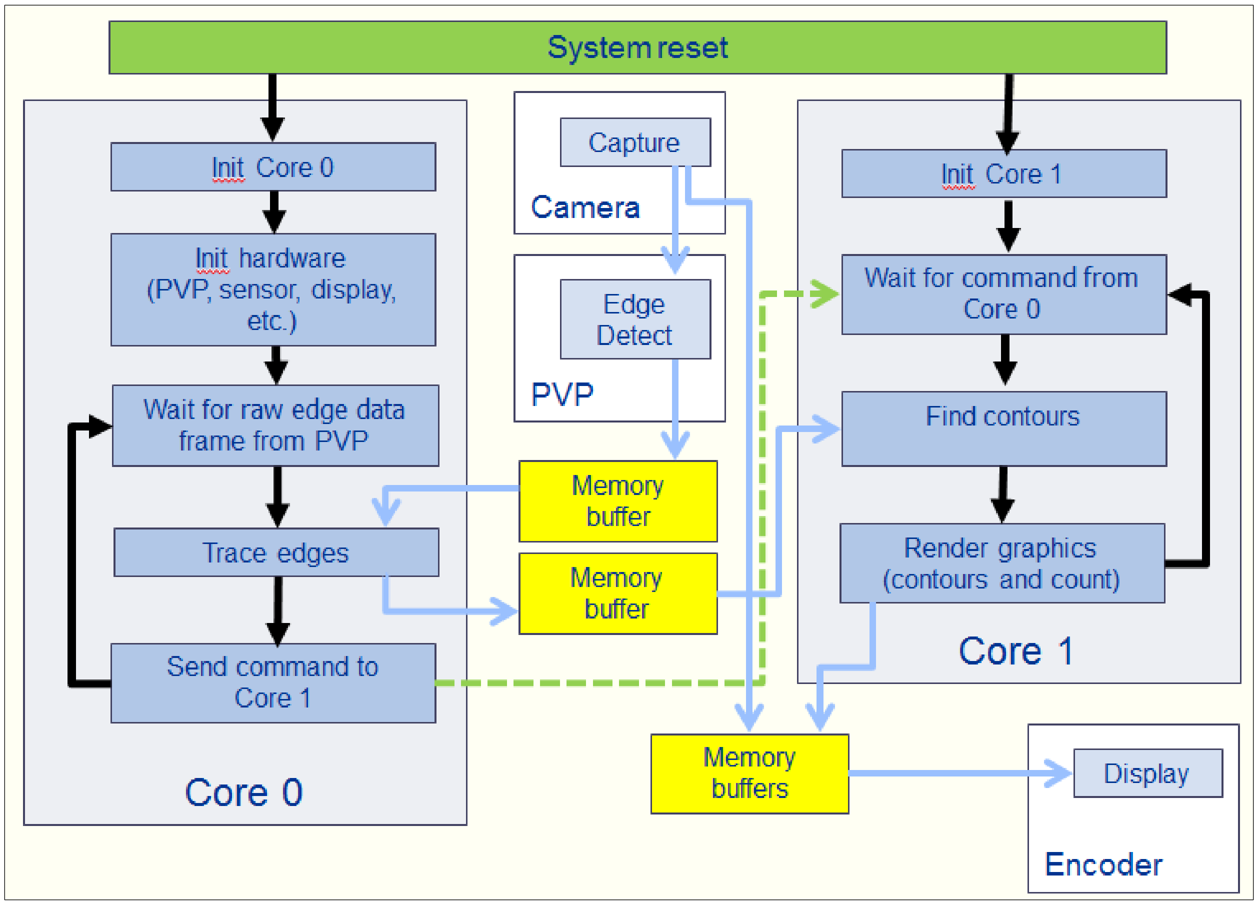


Figure 21. The Distribution Across Cores for the Edge Example

4.2.2 Core 0 Control Flow Graph

Figure 22 illustrates the call graph for Core 0. The entry point for Core 0 is `main()` in `EdgeDetection.c`, which calls the initialization code for the following subfunctions: PVP, multi-core communications, power supply, clock generator, GPIO, HB LED driver, and the image sensor. The image sensor is configured in the module `Sensor.c` by the function `ConfigureSensor()`. A table containing constants for all the sensor's I2C registers is located at the top of `Sensors.c`. To alter a parameter in the sensor (for example, AE mode), an I2C register entry in the `ADI_MT9M114_RegConf_480p_38_4MHz[]` table at the top of `Sensors.c` would be modified. Details about the sensor's registers can be found in the guide available from the manufacturer, Aptina.

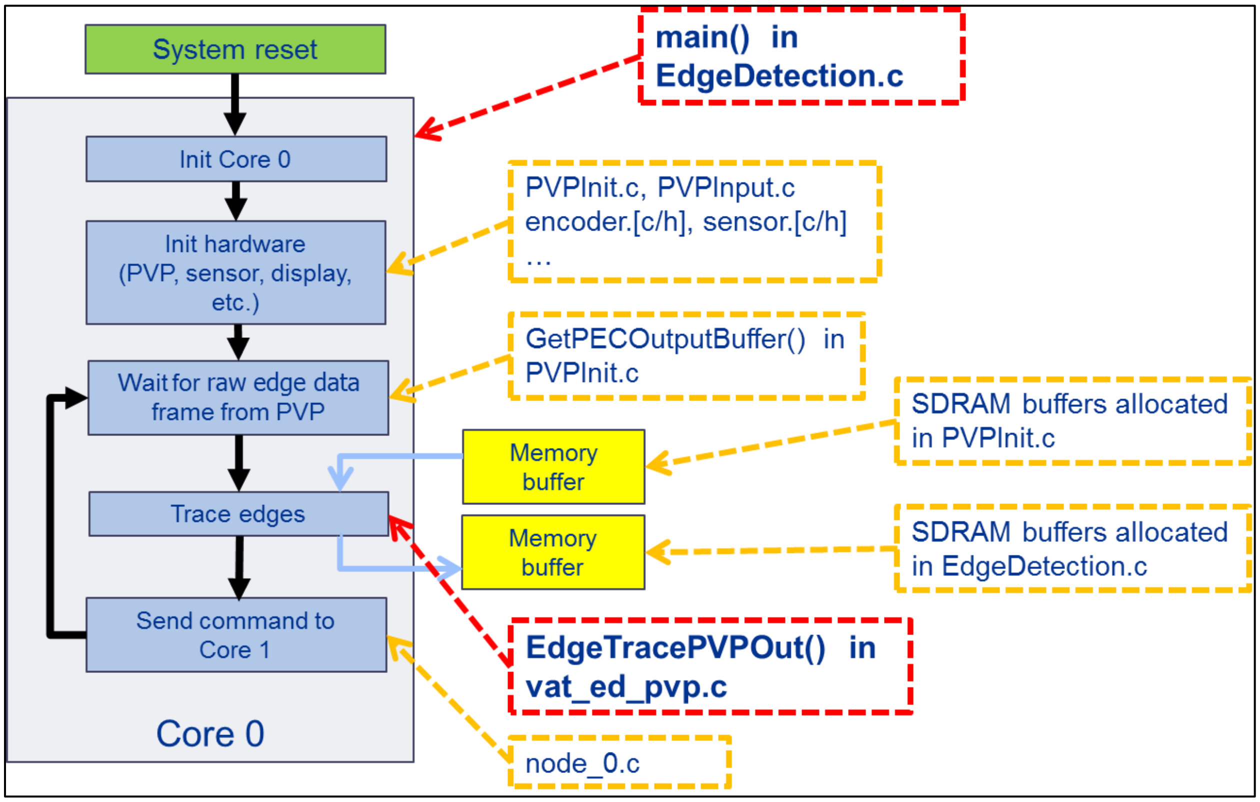
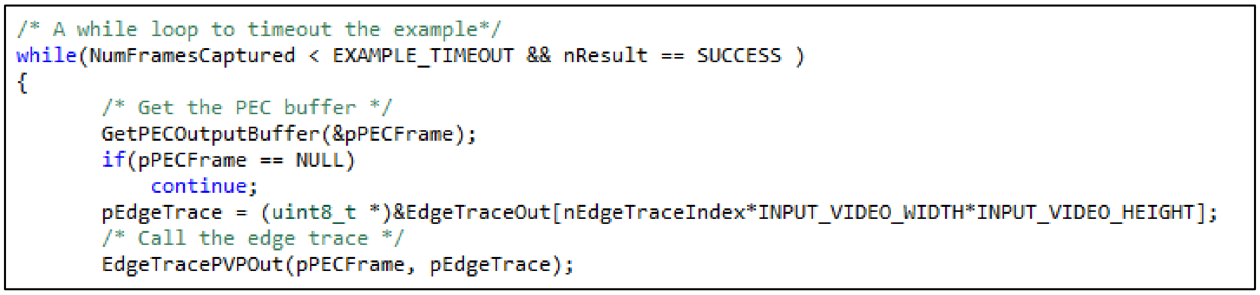

System reset
Init Core 0
Init hardware (PVP, sensor, display, etc.)
Wait for raw edge data frame from PVP
Trace edges
Send command to Core 1
Core 0
main() in EdgeDetection.c
PVPInit.c, PVPInput.c encoder.[c/h], sensor.[c/h] …
GetPECOutputBuffer() in PVPInit.c
Memory buffer
Memory buffer
SDRAM buffers allocated in PVPInit.c
SDRAM buffers allocated in EdgeDetection.c
EdgeTracePVPOut() in vat_ed_pvp.c
node_0.c

Figure 22. Core 0 Control Flow Graph

4.2.2.1 Video Input Loop

The video input loop shuttles PEC buffers from the PVP to the edge trace library and calls the multi-core library to transfer the edge trace buffers to Core 1. The PEC buffer is then freed back to the PEC buffer pool.

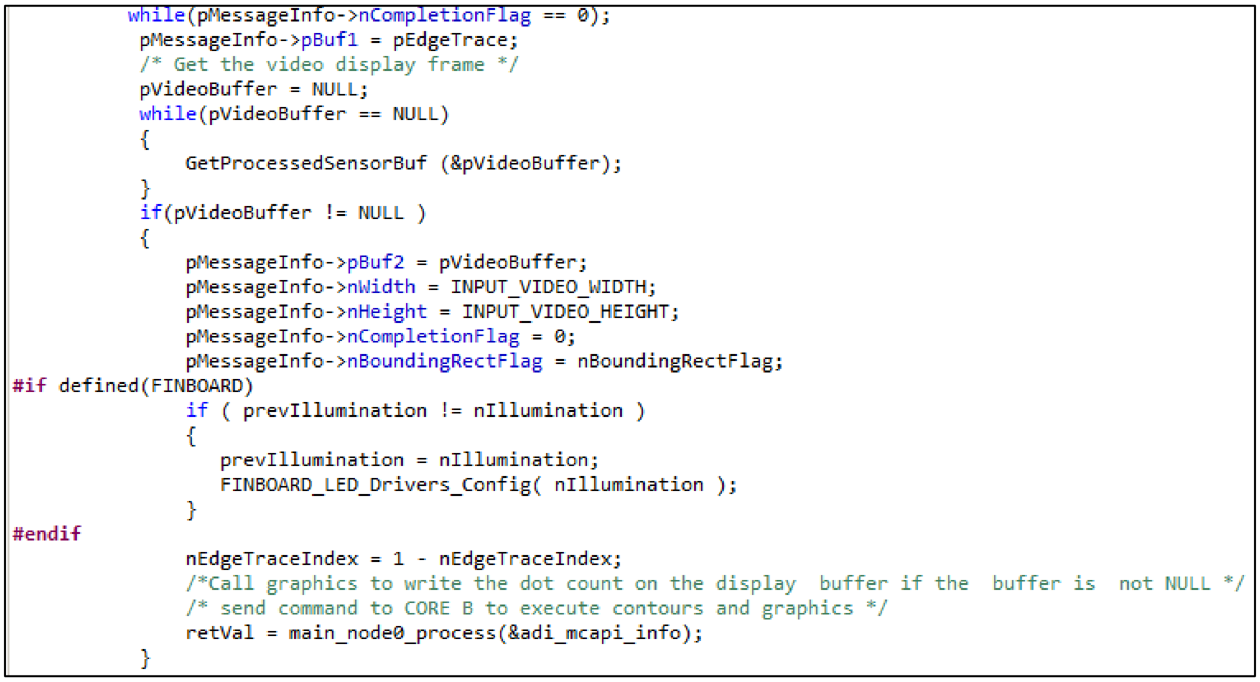
The code snippet in Figure 23 shows how the video input loop gets a full PEC buffer and passes it to the edge trace library. The `GetPECOutputBuffer()` call retrieves a frame from the PEC output buffer pool discussed in Section 4.1.8 (Page 21) and passes the buffer (`pPECFrame`) to the ADI edge trace library. The edge trace library outputs to the buffer `pEdgeTrace`. The EdgeTrace memory pool is independent of both the video and PEC buffer pools.



```
/* A while loop to timeout the example*/
while(NumFramesCaptured < EXAMPLE_TIMEOUT && nResult == SUCCESS )
{
        /* Get the PEC buffer */
        GetPECOutputBuffer(&pPECFrame);
        if(pPECFrame == NULL)
            continue;
        pEdgeTrace = (uint8_t *)&EdgeTraceOut[nEdgeTraceIndex*INPUT_VIDEO_WIDTH*INPUT_VIDEO_HEIGHT];
        /* Call the edge trace */
        EdgeTracePVPOut(pPECFrame, pEdgeTrace);
```

Figure 23. Code Snippet from the Video Input Loop in the Module EdgeDetection.c

Figure 24 is a code snippet showing the next step in the video input loop. The function `GetProcessedSensorBuf()` is located in `Sensor.c`. The function gets a video buffer from the raw camera feed, which is a full color frame used to display the raw video image on the monitor.



```
          while(pMessageInfo->nCompletionFlag == 0);
           pMessageInfo->pBuf1 = pEdgeTrace;
           /* Get the video display frame */
           pVideoBuffer = NULL;
           while(pVideoBuffer == NULL)
           {
               GetProcessedSensorBuf (&pVideoBuffer);
           }
           if(pVideoBuffer != NULL )
           {
               pMessageInfo->pBuf2 = pVideoBuffer;
               pMessageInfo->nWidth = INPUT_VIDEO_WIDTH;
               pMessageInfo->nHeight = INPUT_VIDEO_HEIGHT;
               pMessageInfo->nCompletionFlag = 0;
               pMessageInfo->nBoundingRectFlag = nBoundingRectFlag;
#if defined(FINBOARD)
               if ( prevIllumination != nIllumination )
               {
                  prevIllumination = nIllumination;
                  FINBOARD_LED_Drivers_Config( nIllumination );
               }
#endif
               nEdgeTraceIndex = 1 - nEdgeTraceIndex;
               /*Call graphics to write the dot count on the display  buffer if the  buffer is  not NULL */
               /* send command to CORE B to execute contours and graphics */
               retVal = main_node0_process(&adi_mcapi_info);
           }
```

Figure 24. Code Snippet from the Video Input Loop in the Module EdgeDetection.c

Pointers to the `EdgeTrace` buffer and the VideoBuffer are packaged into the `MessageInfo` structure. The `EdgeBuffer` pointer is assigned to `pBuf1` and the `VideoBuffer` pointer is assigned to `pBuf2`. The pointer `pMessageInfo` is mapped earlier in `EdgeDetection.c` to a global memory region called `buffer[]`, which is declared in `node_0.c`. The data in `buffer[]` is passed between cores by the ADI MCAPI library in the module `node_0.c`. The video buffers are shared memory; only the video buffer pointer is sent to Core 1.

4.2.2.2 Displaying an Image on the Screen

Figure 25 is a picture of the demo output on an HDMI monitor. The demo outputs a 1280×720 image, while the camera input is 720×480. The raw 720×480 color video from the camera is shown on the left, with a graphic overlay (the red dots). In the background is a static image and along the bottom are the dot count and software version.

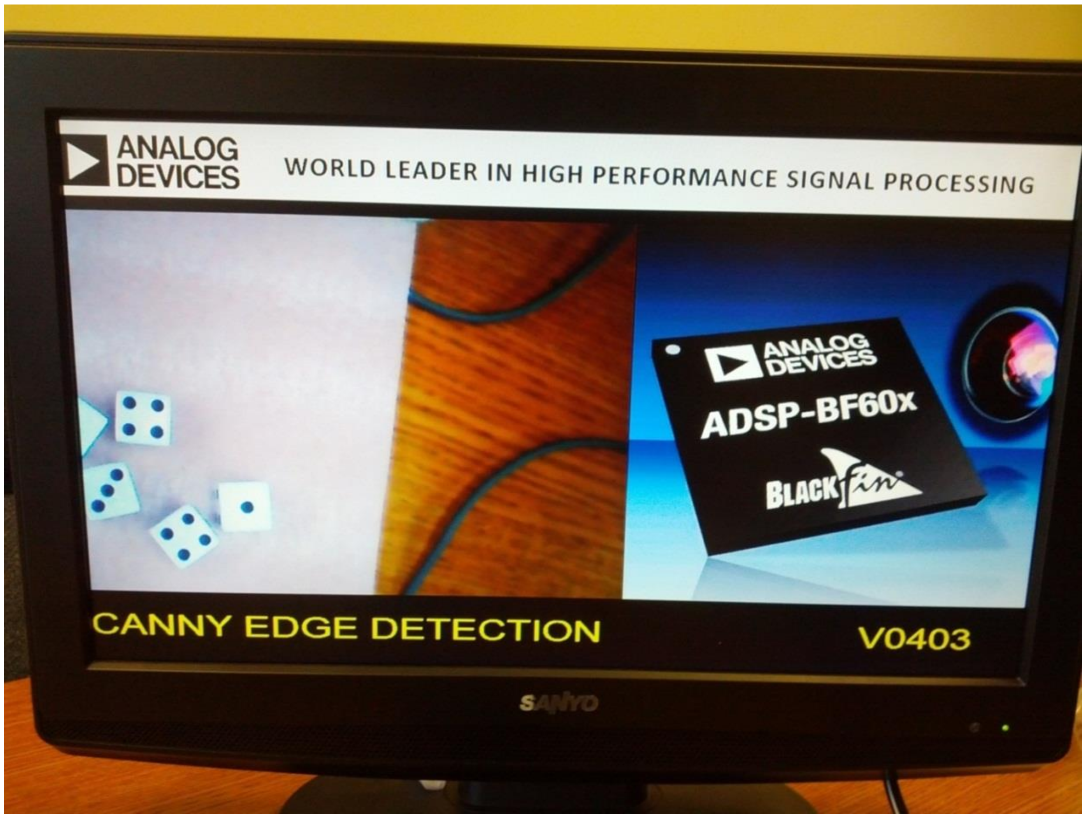


Figure 25. Canny Edge Detection Framework Output

The design uses an ADV7511 driver located in the file `encoder.c` running on Core 0 to output video. For every output video frame, the function `VideoEncAdv7511Callback()` is called by the driver to get the next 1280×720 video buffer ready to display. Both the camera and the display driver share the same video buffer pool. The video buffer pool is initialized in `Sensor.c` and the 1280×720 static buffer image (located in `image.c`) is copied into each buffer. The static image copy is only done once during initialization.

The camera driver is configured to insert the raw video frame into the larger output video frame by setting a stride in the DMA controller. This is done on Core 0, and the video buffers are passed to Core 1 by the video input loop in `EdgeDetection.c`. The video buffers are shared memory, with only the video buffer pointer sent to Core 1.

On Core 1 the OpenGL based graphic library calls `GetFrameBuffer()` in `Graphics.c` to get the next surface to draw on. `GetFrameBuffer()` returns `pInfo→pBuf2`. As discussed in Section 4.2.2.1 (Page 25) `pBuf2` is set in Core 0 to the current video buffer. The OpenGL graphics library draws on the video buffer and returns. In `node_1.c` the video buffer is marked as ready to display. The ADV7511 display driver sees the video buffer is marked for display, and displays it. After the video buffer is displayed, it is marked as clean. The camera driver sees the clean video buffer and the whole process starts over again.

4.2.2.3 The Life of a Video Buffer

Video buffers are special buffers in the ADI Edge Detection example. They are used to shuttle raw color frames from the camera driver to the graphics library, and then to the display driver as shown in Figure 26. Both the camera driver and the display driver run on Core 0, while the graphics library runs on Core 1. The video buffers are shared memory between the two cores.

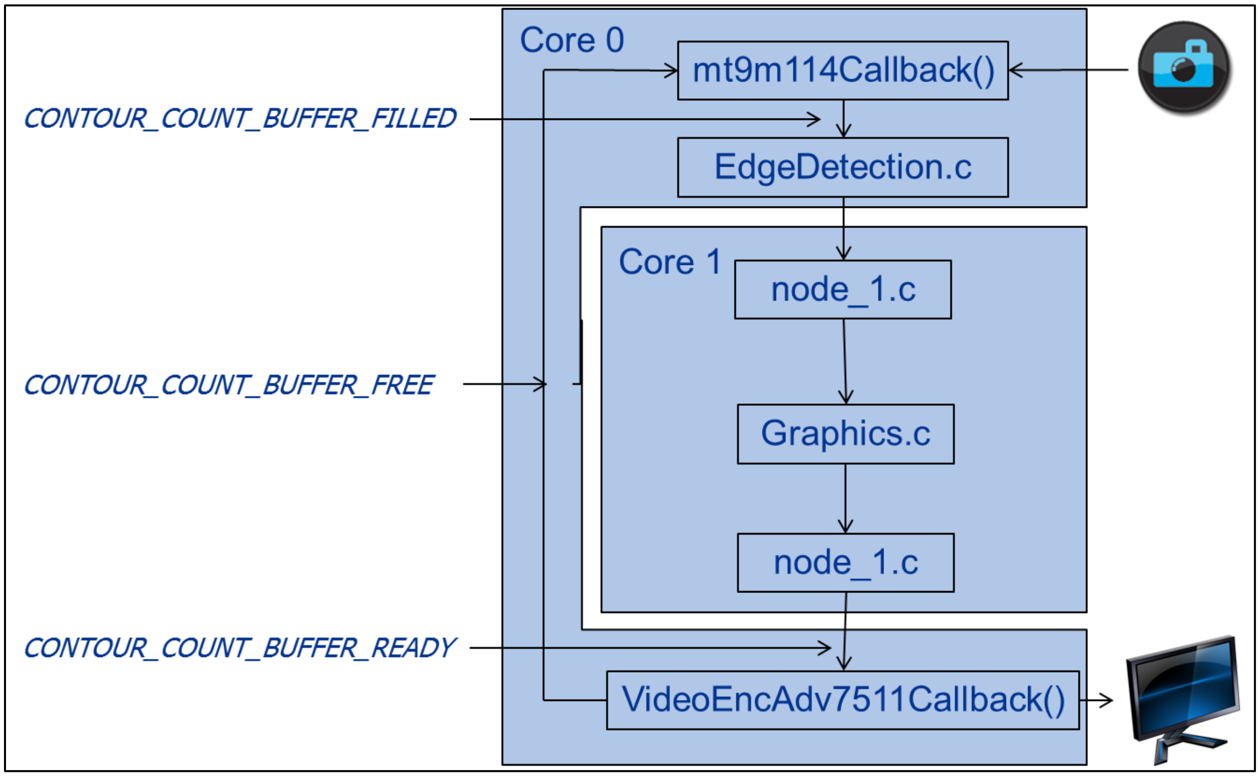
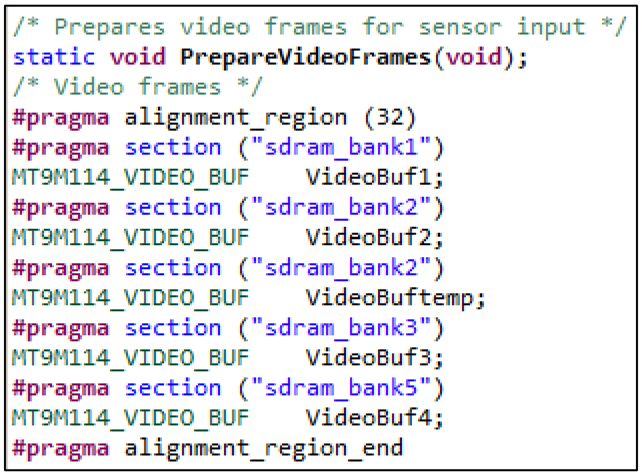

Core 0
mt9m114Callback()
CONTOUR_COUNT_BUFFER_FILLED
EdgeDetection.c
Core 1
node_1.c
CONTOUR_COUNT_BUFFER_FREE
Graphics.c
node_1.c
CONTOUR_COUNT_BUFFER_READY
VideoEncAdv7511Callback()

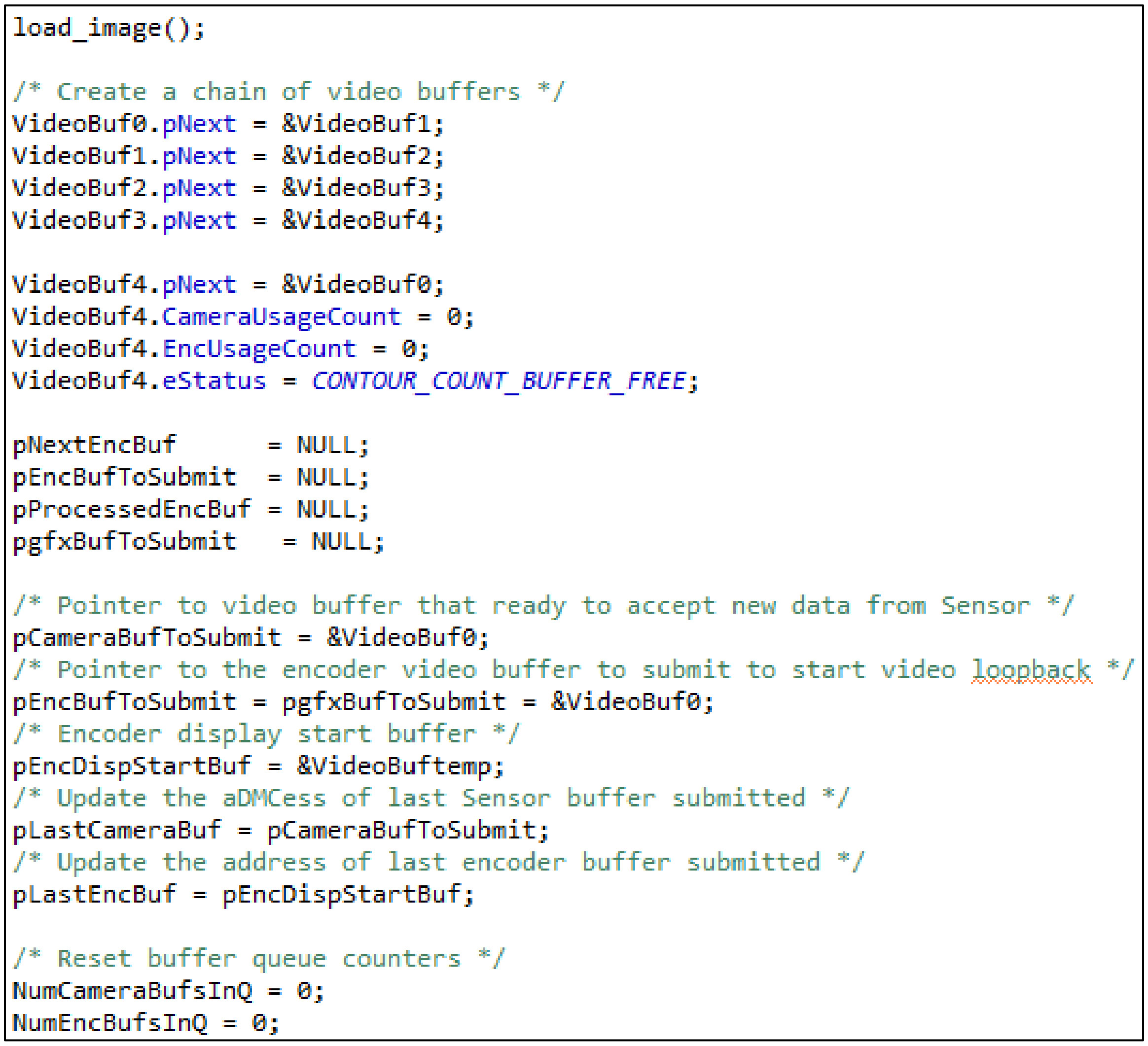
Figure 26. The Life of a Video Buffer

The code snippet in Figure 27 shows the video buffer declarations in `Sensor.c`. Each video buffer is in a separate DRAM bank to take full advantage of DRAM bandwidth. The function `PrepareVideoFrames()` is also in `Sensor.c`. It sets the status of each buffer to `CONTOUR_COUNT_BUFFER_FREE`, loads the static image into each buffer by calling load_`image()`, builds the buffer chain, and finally initializes the buffer pointers. Code snippets from the function `PrepareVideoFrames()` are shown in Figure 28.



```
/* Prepares video frames for sensor input */
static void PrepareVideoFrames(void);
/* Video frames */
#pragma alignment_region (32)
#pragma section ("sdram_bank1")
MT9M114_VIDEO_BUF    VideoBuf1;
#pragma section ("sdram_bank2")
MT9M114_VIDEO_BUF    VideoBuf2;
#pragma section ("sdram_bank2")
MT9M114_VIDEO_BUF    VideoBuftemp;
#pragma section ("sdram_bank3")
MT9M114_VIDEO_BUF    VideoBuf3;
#pragma section ("sdram_bank5")
MT9M114_VIDEO_BUF    VideoBuf4;
#pragma alignment_region_end
```

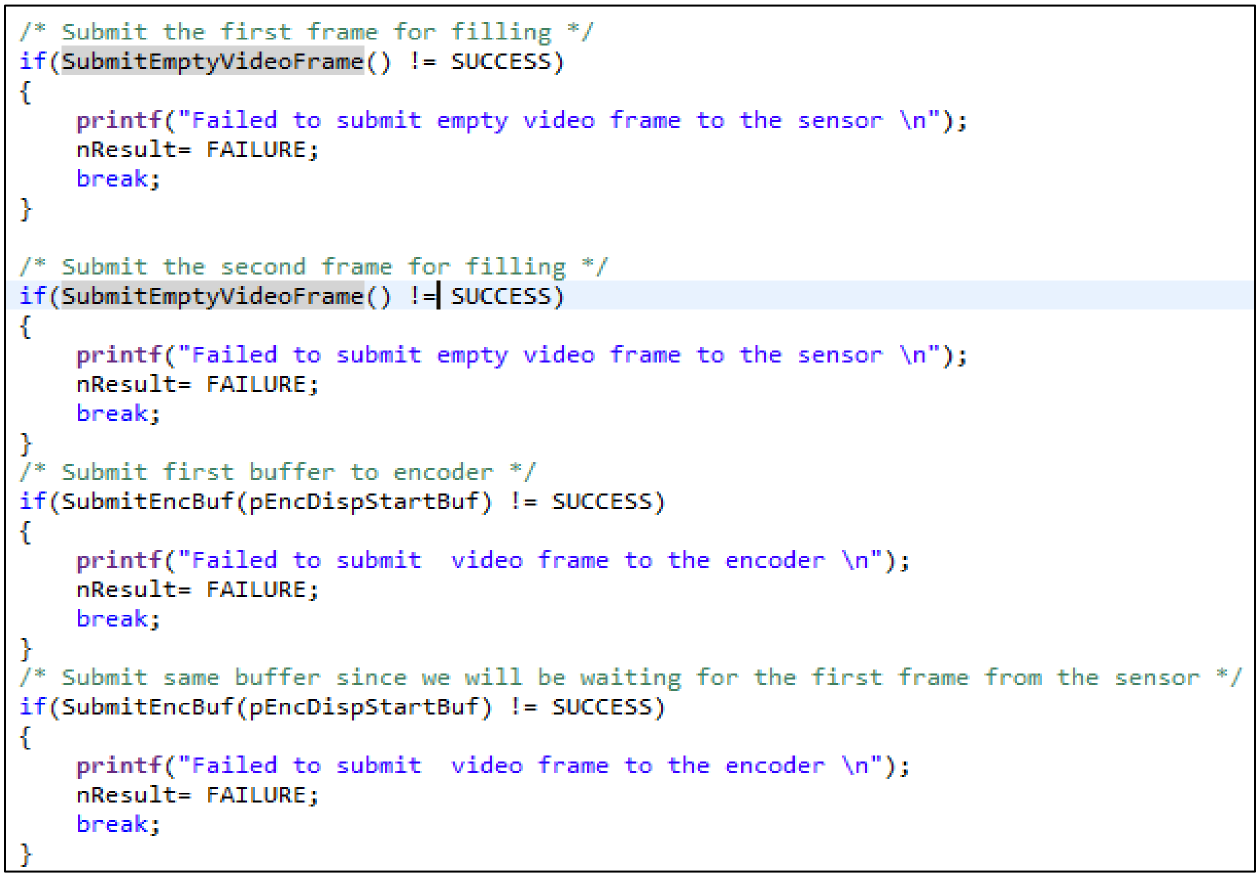
Figure 27. The Video Buffers are Declared in Sensor.c



```
load_image();

/* Create a chain of video buffers */
VideoBuf0.pNext = &VideoBuf1;
VideoBuf1.pNext = &VideoBuf2;
VideoBuf2.pNext = &VideoBuf3;
VideoBuf3.pNext = &VideoBuf4;

VideoBuf4.pNext = &VideoBuf0;
VideoBuf4.CameraUsageCount = 0;
VideoBuf4.EncUsageCount = 0;
VideoBuf4.eStatus = CONTOUR_COUNT_BUFFER_FREE;

pNextEncBuf      = NULL;
pEncBufToSubmit  = NULL;
pProcessedEncBuf = NULL;
pgfxBufToSubmit   = NULL;

/* Pointer to video buffer that ready to accept new data from Sensor */
pCameraBufToSubmit = &VideoBuf0;
/* Pointer to the encoder video buffer to submit to start video loopback */
pEncBufToSubmit = pgfxBufToSubmit = &VideoBuf0;
/* Encoder display start buffer */
pEncDispStartBuf = &VideoBuftemp;
/* Update the aDMCess of last Sensor buffer submitted */
pLastCameraBuf = pCameraBufToSubmit;
/* Update the address of last encoder buffer submitted */
pLastEncBuf = pEncDispStartBuf;

/* Reset buffer queue counters */
NumCameraBufsInQ = 0;
NumEncBufsInQ = 0;
```

Figure 28. Code Snippets from the Function PrepareVideoFrames() in Sensor.c

The first video buffers are submitted in the initialization portion of `EdgeDetection.c`. Two video buffers are submitted to the sensor and two video buffers are submitted to the encoder (as shown in Figure 29).



```
/* Submit the first frame for filling */
if(SubmitEmptyVideoFrame() != SUCCESS)
{
    printf("Failed to submit empty video frame to the sensor \n");
    nResult= FAILURE;
    break;
}

/* Submit the second frame for filling */
if(SubmitEmptyVideoFrame() != SUCCESS)
{
    printf("Failed to submit empty video frame to the sensor \n");
    nResult= FAILURE;
    break;
}
/* Submit first buffer to encoder */
if(SubmitEncBuf(pEncDispStartBuf) != SUCCESS)
{
    printf("Failed to submit  video frame to the encoder \n");
    nResult= FAILURE;
    break;
}
/* Submit same buffer since we will be waiting for the first frame from the sensor */
if(SubmitEncBuf(pEncDispStartBuf) != SUCCESS)
{
    printf("Failed to submit  video frame to the encoder \n");
    nResult= FAILURE;
    break;
}
```

Figure 29. Code Snippet from EdgeDetection.c showing the "Priming of the Pump"

When the sensor is enabled it will transfer DMA video into the video buffer, then via interrupt call the sensor callback function—`mt9m114Callback()` in `Sensor.c`. `Mt9m114Callback()` marks the filled buffer as `CONTOUR_COUNT_BUFFER_FILLED` and calls `SubmitSensorBuf()` in `Sensor.c` to submit the next available buffer marked `CONTOUR_COUNT_BUFFER_FREE`.

While the sensor is filling video buffers under interrupt control (asynchronous to the rest of the system), the video input loop in `EdgeDetection.c` is calling the function `GetProcessedSensorBuf()`. This function is located in `Sensor.c` and returns a pointer to the next video buffer in the chain marked as `CONTOUR_COUNT_BUFFER_FILLED`. As described above, video buffers marked as `CONTOUR_COUNT_BUFFER_FILLED` have been filled with video data from the sensor. The input video loop in `EdgeDetection.c` sends the video buffer pointer returned by the `GetProcessedSensorBuf()` function to Core 1.

In Core 1, the video buffer pointer is picked up by the `GetFrameBuffer()` function in `Graphics.c` as described in Section 4.2.2.2 (Page 26). The OpenGL library uses the video buffer as a surface on which it overlays graphics. After the graphics library is done with the video buffer, the buffer is marked as `CONTOUR_COUNT_BUFFER_READY` in the function `dsp_command_loop()`, case `DSP_GFXCONTOUR`, in the file `node_1.c`.

While all the above is happening, the encoder (display driver) is running asynchronously. As described in Section 4.2.2.2 (Page 26), after a frame is displayed, the `VideoEncAdv7511Callback()` function is called from an interrupt. The callback marks the video buffer it just displayed as `CONTOUR_COUNT_BUFFER_FREE`, then calls the function `GetProcessedGfxBuf()` located in `Sensor.c` to retrieve the next video buffer in the chain marked as `CONTOUR_COUNT_BUFFER_READY`.

When the display driver is done with the video buffer, the buffer is marked as `CONTOUR_COUNT_BUFFER_FREE`. This allows the buffer to be picked up by the `SubmitSensorBuf()` function call in the sensor callback, starting the process over again.

4.2.3 Core 1 Control Flow Graph

Figure 30 illustrates the call graph for Core 1. The purpose of the code running on Core 1 is to find contours and overlay graphics onto video buffers. The entry point for Core 1 is `main()` in `bf609_mcapi_msg_core.c`. `Main()` initializes the graphics library and calls `main_node1()` in module `node_1.c`. The function `main_node1()` calls `dsp_command_loop()`, which is the primary loop for Core 1.

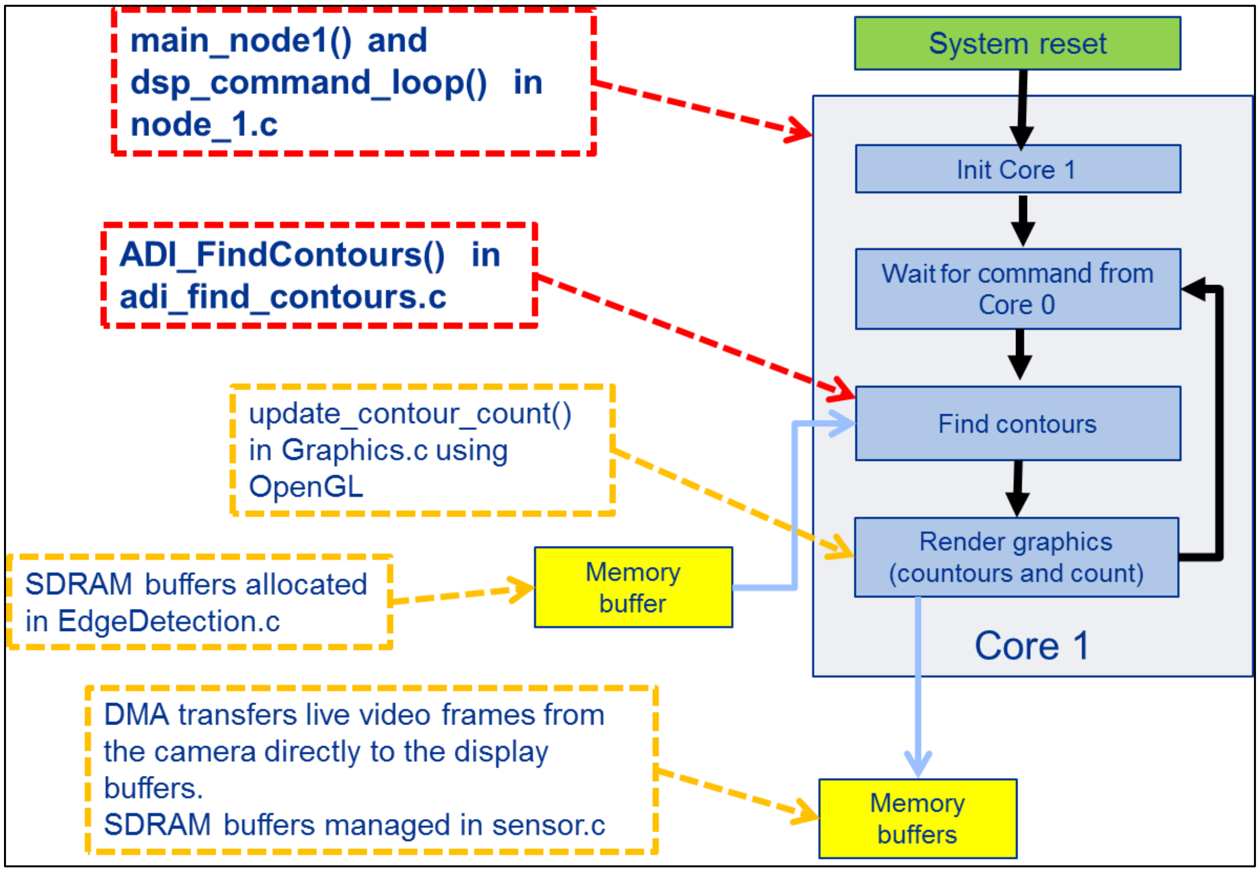
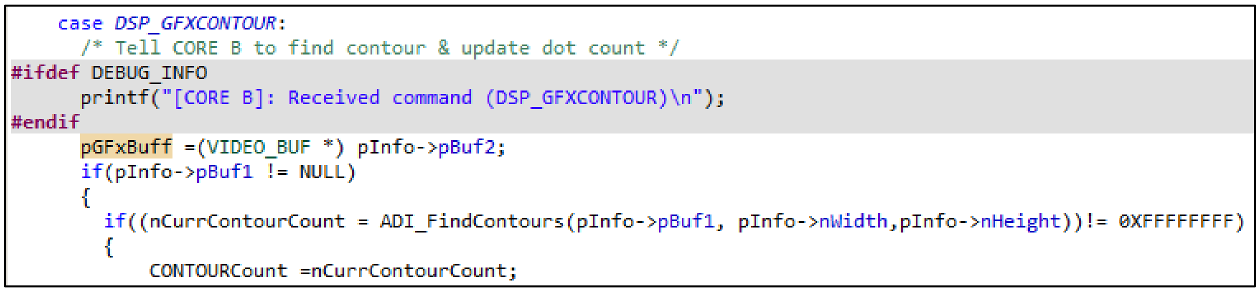

main_node1() and dsp_command_loop() in node_1.c
ADI_FindContours() in adi_find_contours.c
update_contour_count() in Graphics.c using OpenGL
SDRAM buffers allocated in EdgeDetection.c
Memory buffer
DMA transfers live video frames from the camera directly to the display buffers.
SDRAM buffers managed in sensor.c
Memory buffers
System reset
Init Core 1
Wait for command from Core 0
Find contours
Render graphics (countours and count)
Core 1

Figure 30. Core 1 Control Flow Graph

4.2.3.1 Core 1 dsp_command_loop()

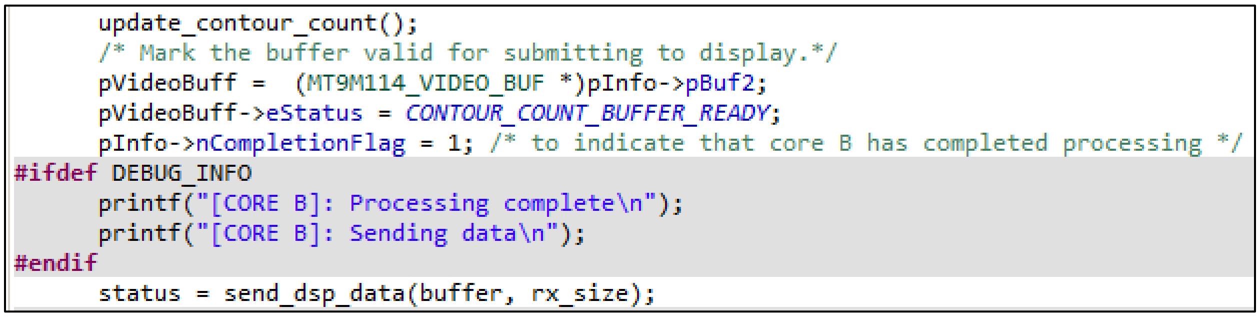
The video input loop in `EdgeDetection.c` gets a video buffer and an edge trace buffer and sends pointers to Core 1 using a `DSP_GFXCONTOUR` command. Running on Core 1 is the `dsp_command_loop()` located in the file `node_1.c`. When the `DSP_GFXCONTOUR` command is received, the edge trace buffer from Core 0 is passed to `ADI_FindContours()`, located in `adi_findcontours.c`. The video buffer pointer is assigned to the global variable `pGFxBuff` as shown in Figure 31.



```
    case DSP_GFXCONTOUR:
      /* Tell CORE B to find contour & update dot count */
#ifdef DEBUG_INFO
      printf("[CORE B]: Received command (DSP_GFXCONTOUR)\n");
#endif
      pGFxBuff =(VIDEO_BUF *) pInfo->pBuf2;
      if(pInfo->pBuf1 != NULL)
      {
        if((nCurrContourCount = ADI_FindContours(pInfo->pBuf1, pInfo->nWidth,pInfo->nHeight))!= 0XFFFFFFFF)
        {
            CONTOURCount =nCurrContourCount;
```

Figure 31. dsp_command_loop() in node_1.c

`ADI_FindContours()` returns a dot count (in the BDTI Dice Dot-Counting Demo and Reference Design) or a contour count (in the ADI Canny Edge Detection Example). As shown in Figure 32, after the dot or contour count is measured, the `update_contour_count()` function in `Graphics.c` is called. This function overlays graphics onto the video buffer pointed to by `pGFxBuff`. After the graphics library is done with the overlay, the video buffer is marked as `CONTOUR_COUNT_BUFFER_READY`, freeing it to be shown by the display driver. Finally the `CompletionFlag` is set and the `pMessageInfo` buffer is sent back to Core 0, where it is used for synchronization in the video input loop.



```
        update_contour_count();
        /* Mark the buffer valid for submitting to display.*/
        pVideoBuff =  (MT9M114_VIDEO_BUF *)pInfo->pBuf2;
        pVideoBuff->eStatus = CONTOUR_COUNT_BUFFER_READY;
        pInfo->nCompletionFlag = 1; /* to indicate that core B has completed processing */
#ifdef DEBUG_INFO
        printf("[CORE B]: Processing complete\n");
        printf("[CORE B]: Sending data\n");
#endif
        status = send_dsp_data(buffer, rx_size);
```

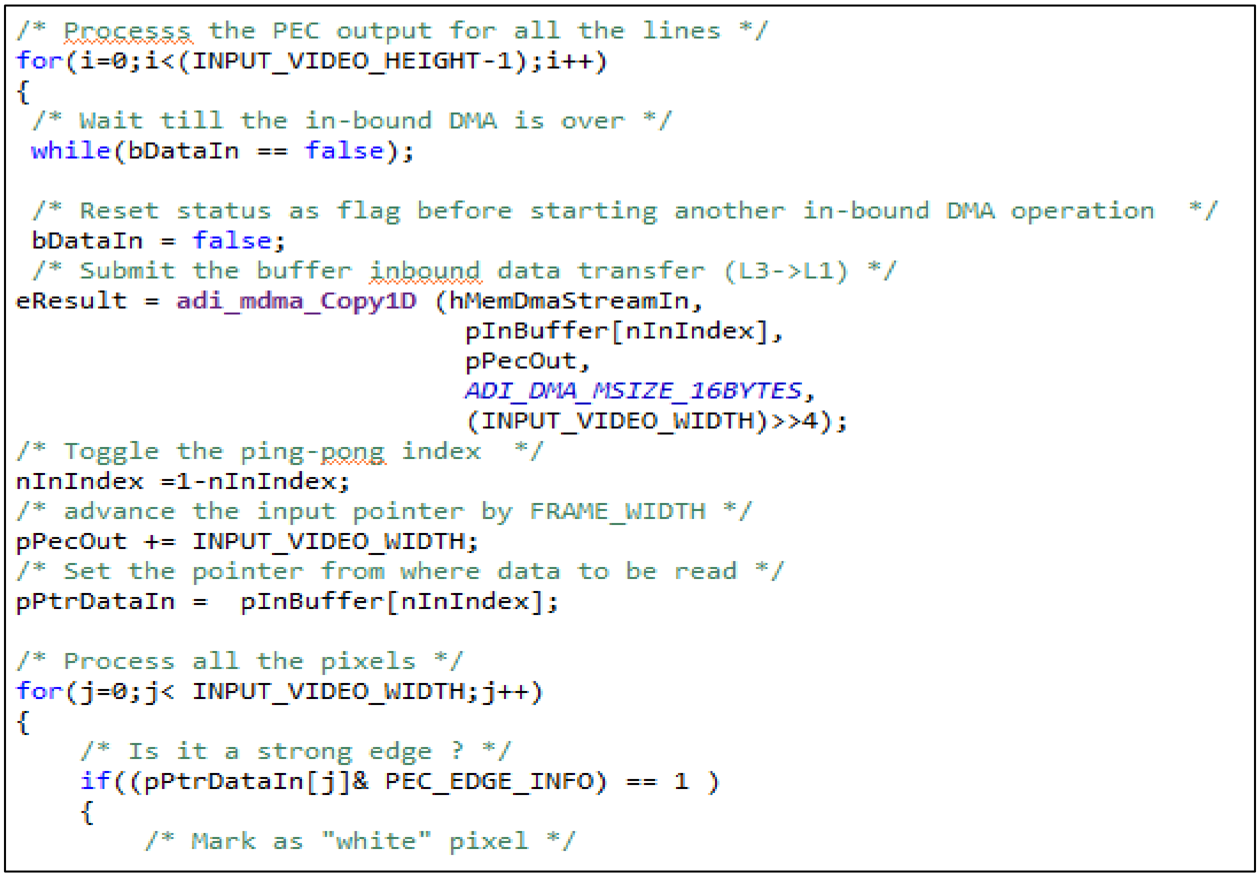
Figure 32. dsp_command_loop() in node_1.c

4.2.3.2 Graphics

Graphics overlay is done using the ADI OpenGL graphics library. Section 4.2.2.3 (Page 27) explains how the static graphics are written to each video buffer during `init` in the function `load_image()` in the file `Sensor.c` running on core 0. The graphic overlay done in `Graphics.c` is the yellow wording at the bottom of the screen, including the label "Dot Count:", the actual dot count, and the demo mode graphics. There are two demo debug modes selected by the variable `nBoundingRectFlag` and passed to Core 1 from the video input loop running on Core 0. When `nBoundingRectFlag` equals 1, bounding rectangles are drawn overlaid on top of the real time video around each contour. When `nBoundingRectFlag` equals 2, contour pixels are shown as red overlays on top of the real-time video.

The function `update_contour_count()` in `Graphics.c` is responsible for all graphic overlays. `Update_contour_count()` calls the function `display_count()` to actually render the graphics. The graphics are rendered on a surface pointed to by `pGFxBuff`. As discussed earlier, this pointer is assigned a video buffer in the `dsp_command_loop()`.

The function `display_count()` in `Graphics.c` first clears a region of the video buffer that contained the old count. This is done by filling a buffer with black pixels and bit blting the buffer over the old count. The new count is then written in its place with an OpenGL `GLUT_ADI_bitmap_string()` call. If `nBoundingRectFlag` equals 1, rectangles are drawn according to the parameters in the `oContourInfo` structure (set in `adi_findcontours.c`). If `nBoundingRectFlag` equals 2, the function `DrawEdge()` in `adi_draw_edge.c` is called. As shown in Figure 33, `DrawEdge()` scans the edge trace buffer for any *strong edge* pixels and overlays a graphic on that pixel in the video buffer.



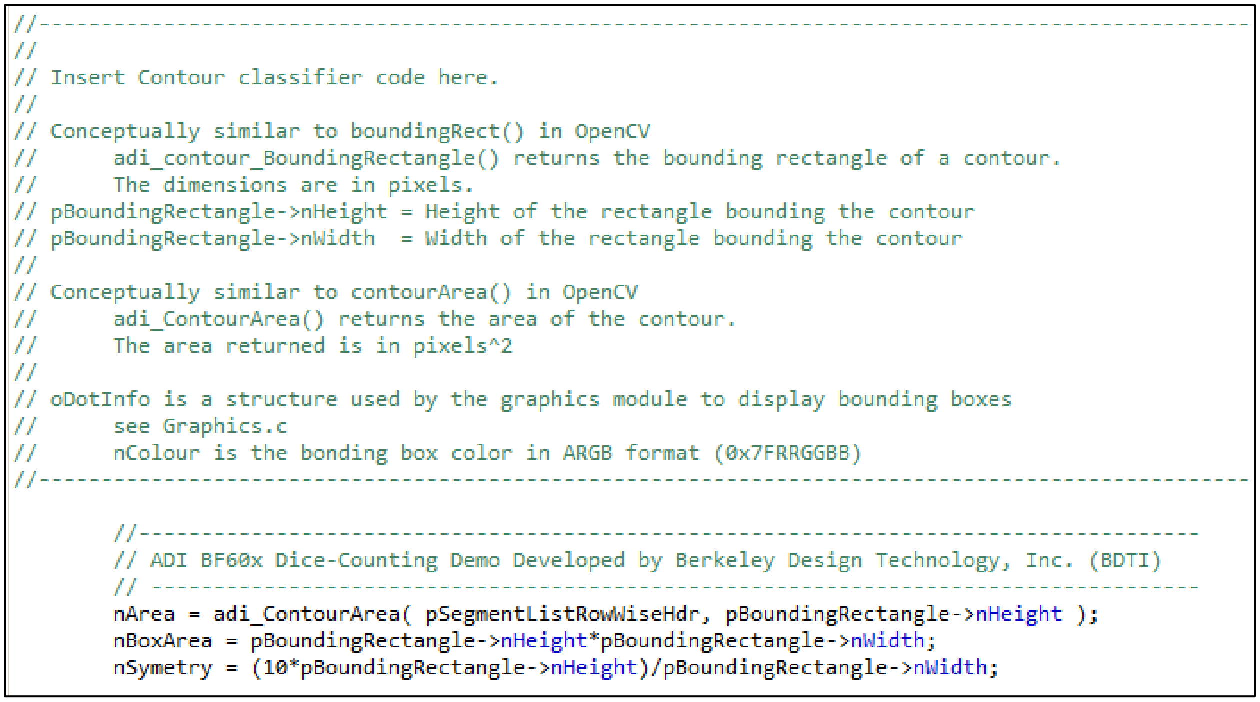
```
/* Processs the PEC output for all the lines */
for(i=0;i<(INPUT_VIDEO_HEIGHT-1);i++)
{
 /* Wait till the in-bound DMA is over */
 while(bDataIn == false);

 /* Reset status as flag before starting another in-bound DMA operation  */
 bDataIn = false;
 /* Submit the buffer inbound data transfer (L3->L1) */
eResult = adi_mdma_Copy1D (hMemDmaStreamIn,
                            pInBuffer[nInIndex],
                            pPecOut,
                            ADI_DMA_MSIZE_16BYTES,
                            (INPUT_VIDEO_WIDTH)>>4);
/* Toggle the ping-pong index  */
nInIndex =1-nInIndex;
/* advance the input pointer by FRAME_WIDTH */
pPecOut += INPUT_VIDEO_WIDTH;
/* Set the pointer from where data to be read */
pPtrDataIn =  pInBuffer[nInIndex];

/* Process all the pixels */
for(j=0;j< INPUT_VIDEO_WIDTH;j++)
{
    /* Is it a strong edge ? */
    if((pPtrDataIn[j]& PEC_EDGE_INFO) == 1 )
    {
        /* Mark as "white" pixel */
```

Figure 33. Code Snippet from DrawEdge in adi_draw_edge.c in Core 1

4.2.3.3 adi_find_contours.c

The module `adi_find_contours.c` is the heart of the ADI Edge Detection example. Inside the function `ADI_FindContours()` is a loop that iterates through each contour. The contour area and bounding rectangle are calculated and made available to a classifier. This is where user classifier code should be inserted, as illustrated in Figure 34.

The ADI find contours library is limited by memory constraints to the maximum number of contours it can manage. The constant `MAX_NUM_RUN_LEN_NODE` in the module `adi_contours.h` sets the maximum number of contours the ADI library will accept. If the maximum number of contours exceeds this constant, all are discarded and the ADI find contours library returns `ADI_ITB_STATUS_FAILURE`.



```
//-----------------------------------------------------------------------------------------------
//
// Insert Contour classifier code here.
//
// Conceptually similar to boundingRect() in OpenCV
//      adi_contour_BoundingRectangle() returns the bounding rectangle of a contour.
//      The dimensions are in pixels.
// pBoundingRectangle->nHeight = Height of the rectangle bounding the contour
// pBoundingRectangle->nWidth  = Width of the rectangle bounding the contour
//
// Conceptually similar to contourArea() in OpenCV
//      adi_ContourArea() returns the area of the contour.
//      The area returned is in pixels^2
//
// oDotInfo is a structure used by the graphics module to display bounding boxes
//      see Graphics.c
//      nColour is the bonding box color in ARGB format (0x7FRRGGBB)
//-----------------------------------------------------------------------------------------------

        //-------------------------------------------------------------------------------------
        // ADI BF60x Dice-Counting Demo Developed by Berkeley Design Technology, Inc. (BDTI)
        // ------------------------------------------------------------------------------------
        nArea = adi_ContourArea( pSegmentListRowWiseHdr, pBoundingRectangle->nHeight );
        nBoxArea = pBoundingRectangle->nHeight*pBoundingRectangle->nWidth;
        nSymetry = (10*pBoundingRectangle->nHeight)/pBoundingRectangle->nWidth;
```

Figure 34. ADI_FindContours() in the file adi_find_contours.c

4.2.3.4 adi_ContourArea()

The `adi_ContourArea()` function measures the area of a contour using a very simple algorithm. The ADI contours library stores contours as run-length encoded nodes, each one representing a group of continuous edges in a horizontal row of pixels. Each node is represented with a structure containing the X value for the start and end pixel, along with the Y value for the row.

The `adi_ContourArea()` function measures an individual contour area by looking for the minimum start value and maximum end value in each row. The minimum value is subtracted from the maximum, giving the number of pixels within the single contour for that row. All rows in the contour are measured this way and the result accumulated. The final accumulated value is returned as the measured area of the contour.

The simple area algorithm used by the `adi_ContourArea()` function works for circles, triangles, squares, and other simple shapes. The algorithm does not work for complex shapes that contain multiple edge crossings in a single row. If a line drawn through the shape crosses more than two edges, the algorithm will fail to accurately calculate the area.

Figure 35 shows the shapes for which `adi_ContourArea()` can accurately measure area and the shapes for which it cannot. Valid shapes have only two edges on any horizontal line, while invalid shapes have more than two edges on any horizontal line. Shapes can be valid in one orientation but not another. For example, the heart is invalid in the orientation shown in Figure 35, but valid if rotated 90 degrees.

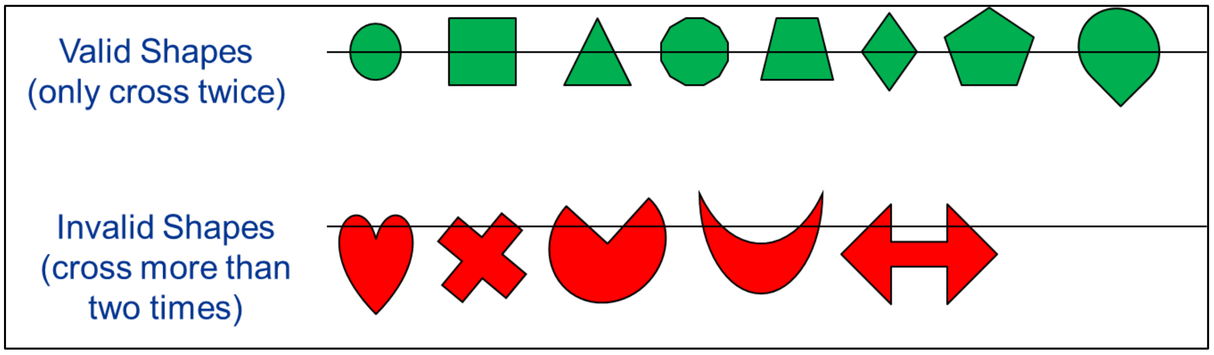


Figure 35. Valid and Invalid Shapes for the adi_ContourArea() Function

Figure 36 is a contour map showing various shapes in various orientations to illustrate a limitation of the area measuring function.

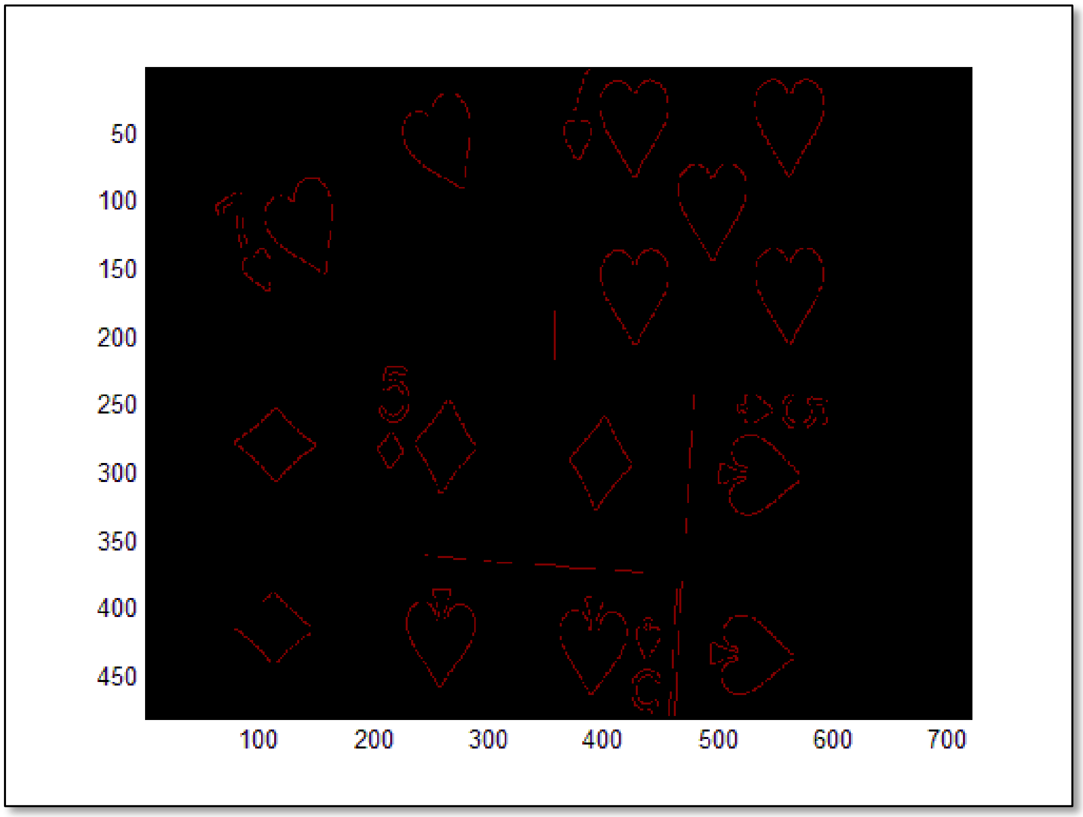


Figure 36. Reference Edges for adi_ContourArea() function Test

Figure 37 shows the results of passing the contours in Figure 36 into the `adi_ContourArea()` function. Red pixels indicate pixels counted as part of the area of the contour. The hearts fail in the upright orientation as expected and the diamonds pass in both orientations as expected. The spades fail in all orientations.

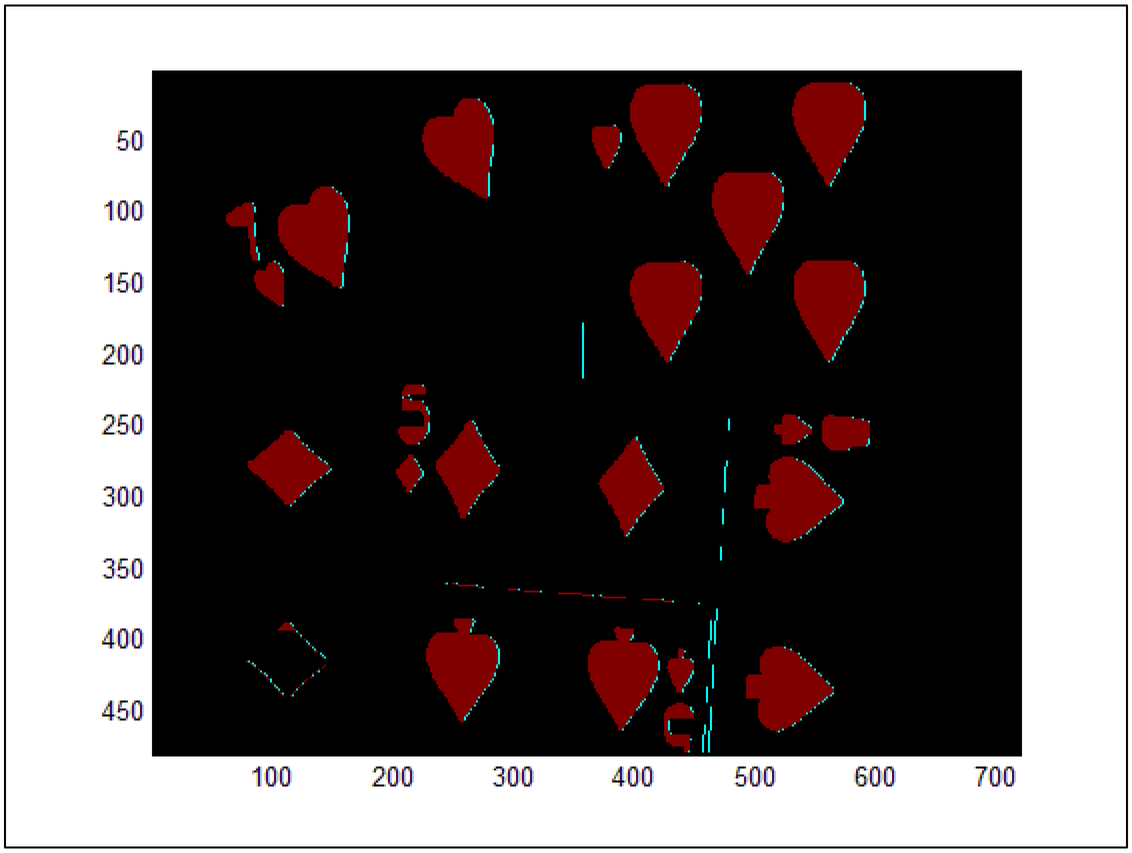


Figure 37. adi_ContourArea() Function Test

4.3 Modifying ADI's Find Contours Library for Dice Dot Classification

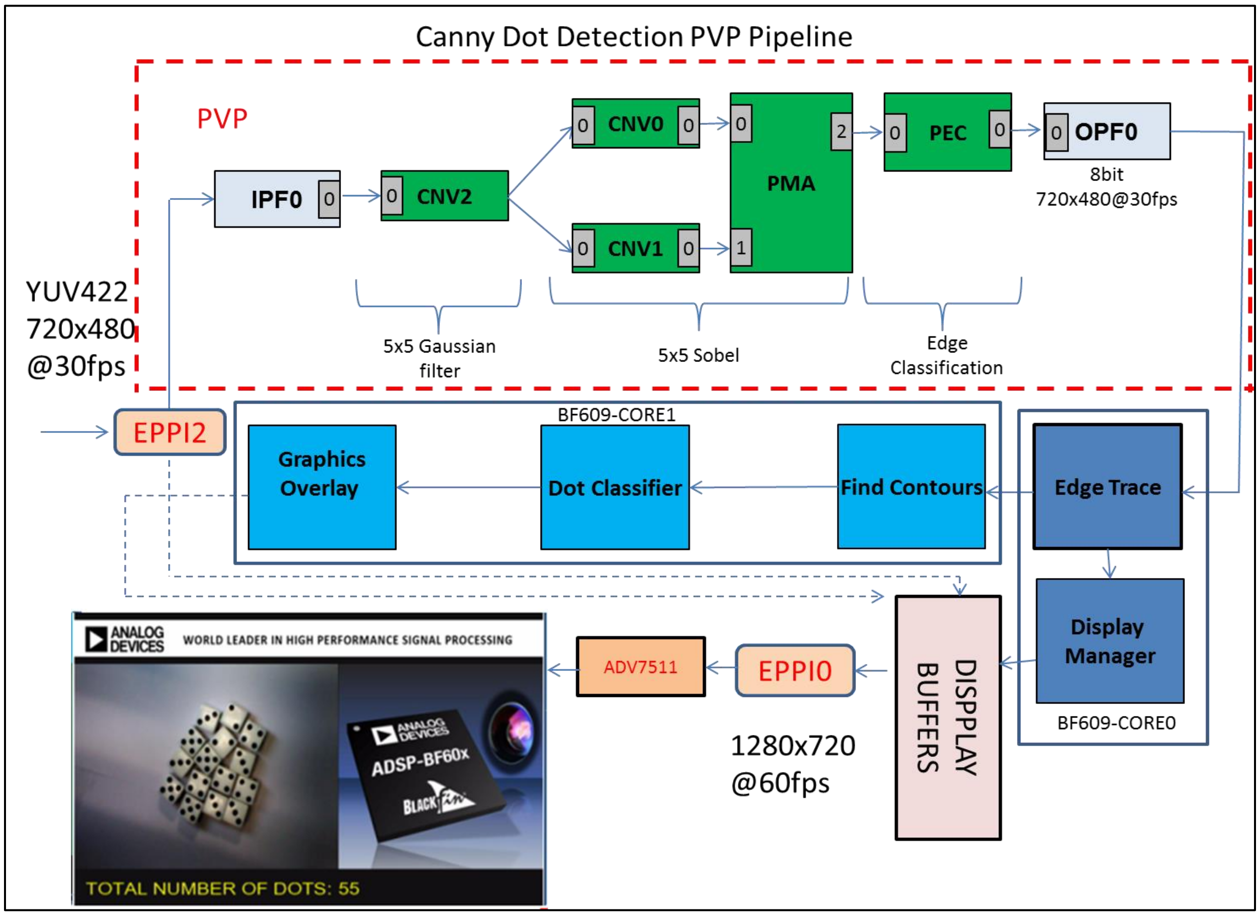


Figure 38. Adding a Dot Classifier to the Edge Detection Demo

4.3.1 ADI Canny Edge Detection Example Files Changed for the Dice Dot-Counting Demo

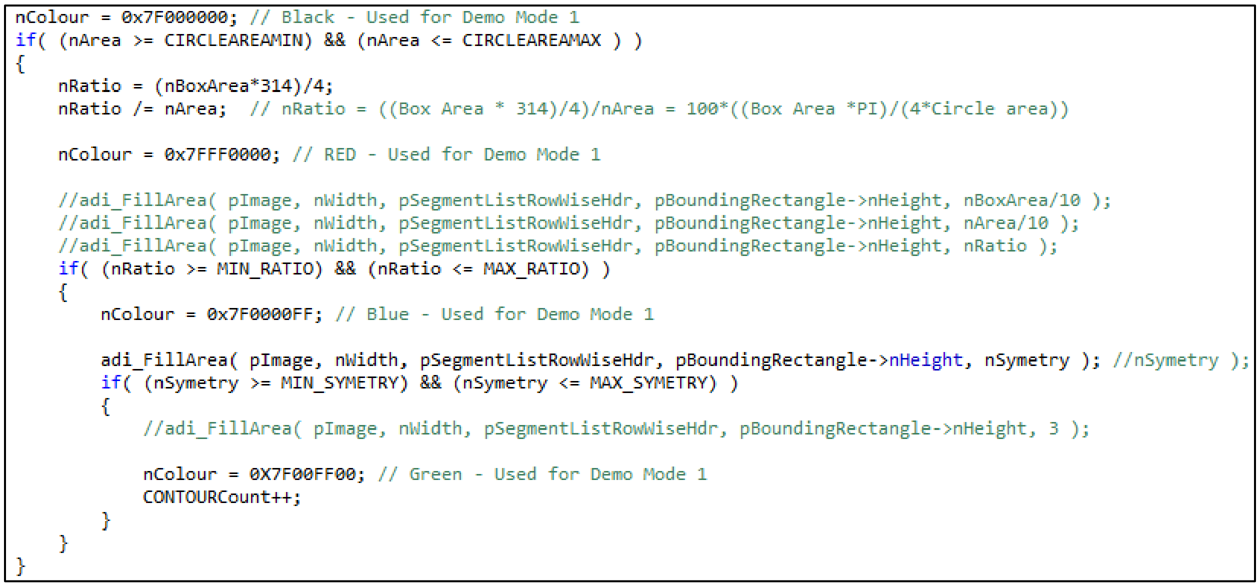
Figure 39 outlines the changes made to the ADI Canny Edge Detection Example to create the BDTI Dice Dot-Counting Demo and Reference Design.

Module	Description of Change
adi_find_contours.c	Added dot classifier
EdgeDetection.c	Added test code
Graphics.c	Added code to display dot count
node_1.c	Added test code
PVPInput.c	Changed PEC thresholds

Figure 39. ADI Canny Edge Detection Example Files Modified for the BDTI Dice Dot-Counting Demo and Reference Design

4.3.2 Adding the Dice Dot Classifier Code

The dice dot classifier is the essence of the BDTI Dice Dot-Counting Demo and Reference Design. The code (shown in Figure 40) is located in `adi_find_contours.c` running on Core 1. The code implements the classifier shown in Figure 8. The classifier itself requires only six lines of code. The additional lines are for debugging and display purposes.

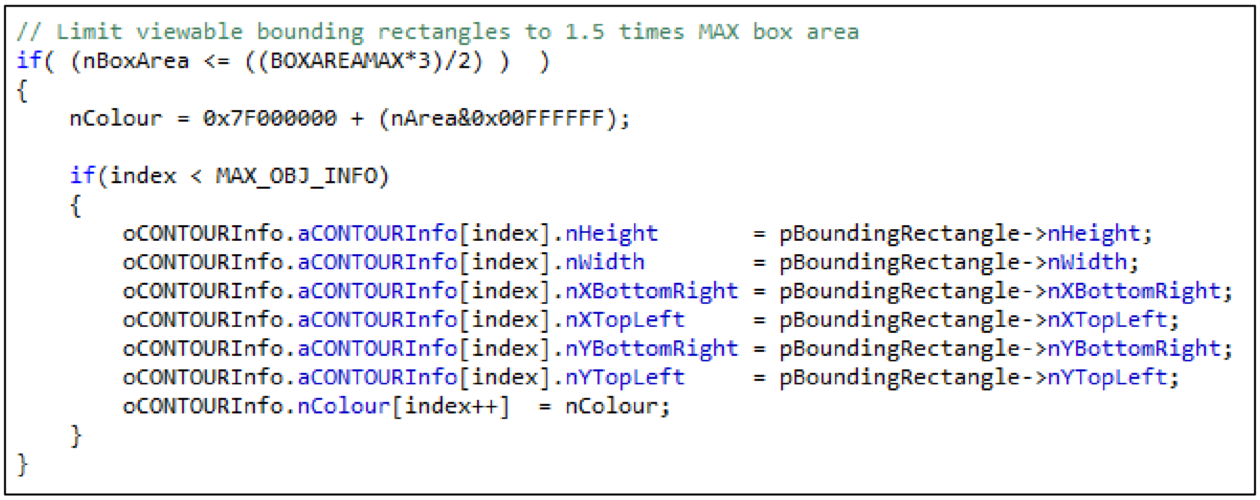


```
nColour = 0x7F000000; // Black - Used for Demo Mode 1
if( (nArea >= CIRCLEAREAMIN) && (nArea <= CIRCLEAREAMAX ) )
{
    nRatio = (nBoxArea*314)/4;
    nRatio /= nArea;  // nRatio = ((Box Area * 314)/4)/nArea = 100*((Box Area *PI)/(4*Circle area))

    nColour = 0x7FFF0000; // RED - Used for Demo Mode 1

    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nBoxArea/10 );
    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nArea/10 );
    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nRatio );
    if( (nRatio >= MIN_RATIO) && (nRatio <= MAX_RATIO) )
    {
        nColour = 0x7F0000FF; // Blue - Used for Demo Mode 1

        adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nSymetry ); //nSymetry );
        if( (nSymetry >= MIN_SYMETRY) && (nSymetry <= MAX_SYMETRY) )
        {
            //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, 3 );

            nColour = 0X7F00FF00; // Green - Used for Demo Mode 1
            CONTOURCount++;
        }
    }
}
```

Figure 40. The Dice Dot Classifier

4.3.3 Displaying Classifier Information in Real-time

The ADI Canny Edge Detection Example code includes support for two real-time debugging modes. The mode is selected using the center button on the FinBoard. In graphics debug mode 1, boxes are drawn around contours and overlaid on the real-time video. In graphics debug mode 2, edge pixels are overlaid on to the real-time video. For mode 1 the color of the box drawn around the contour can be changed depending on characteristics of the contour. An example of how to set the color of a box for mode 1 is shown in Figure 41, where the box color is set according to the area of the contour.



```
// Limit viewable bounding rectangles to 1.5 times MAX box area
if( (nBoxArea <= ((BOXAREAMAX*3)/2) )  )
{
    nColour = 0x7F000000 + (nArea&0x00FFFFFF);

    if(index < MAX_OBJ_INFO)
    {
        oCONTOURInfo.aCONTOURInfo[index].nHeight       = pBoundingRectangle->nHeight;
        oCONTOURInfo.aCONTOURInfo[index].nWidth        = pBoundingRectangle->nWidth;
        oCONTOURInfo.aCONTOURInfo[index].nXBottomRight = pBoundingRectangle->nXBottomRight;
        oCONTOURInfo.aCONTOURInfo[index].nXTopLeft     = pBoundingRectangle->nXTopLeft;
        oCONTOURInfo.aCONTOURInfo[index].nYBottomRight = pBoundingRectangle->nYBottomRight;
        oCONTOURInfo.aCONTOURInfo[index].nYTopLeft     = pBoundingRectangle->nYTopLeft;
        oCONTOURInfo.nColour[index++]   = nColour;
    }
}
```

Figure 41. Debug Mode 1 Structure

4.4 The Dot-Counting Algorithm in Action

Figure 42 through Figure 47 show data captured from various locations in the dot count demo code. Figure 42 and Figure 43 were captured from Core 0 and show raw PEC and edge trace data. Note the color coding in Figure 42: red highlights a *strong edge*, while blue shows a *weak edge*. Notice that in Figure 43 all the *weak edges* are gone and only the *strong edges* remain. The edge trace function is described in section 4.1.4 (Page 16).

Figure 44 through Figure 47 were taken from Core 1. These images show data as it passes through the dot classifier. The progression from Figure 44 to Figure 47 shows the dot classifier discarding contours based on size and shape. Figure 48 shows the final result displayed on a monitor using debug mode 2 (`DrawEdge()`). Note that the red graphics overlay represents contour pixels and dots that have passed through the classifier.

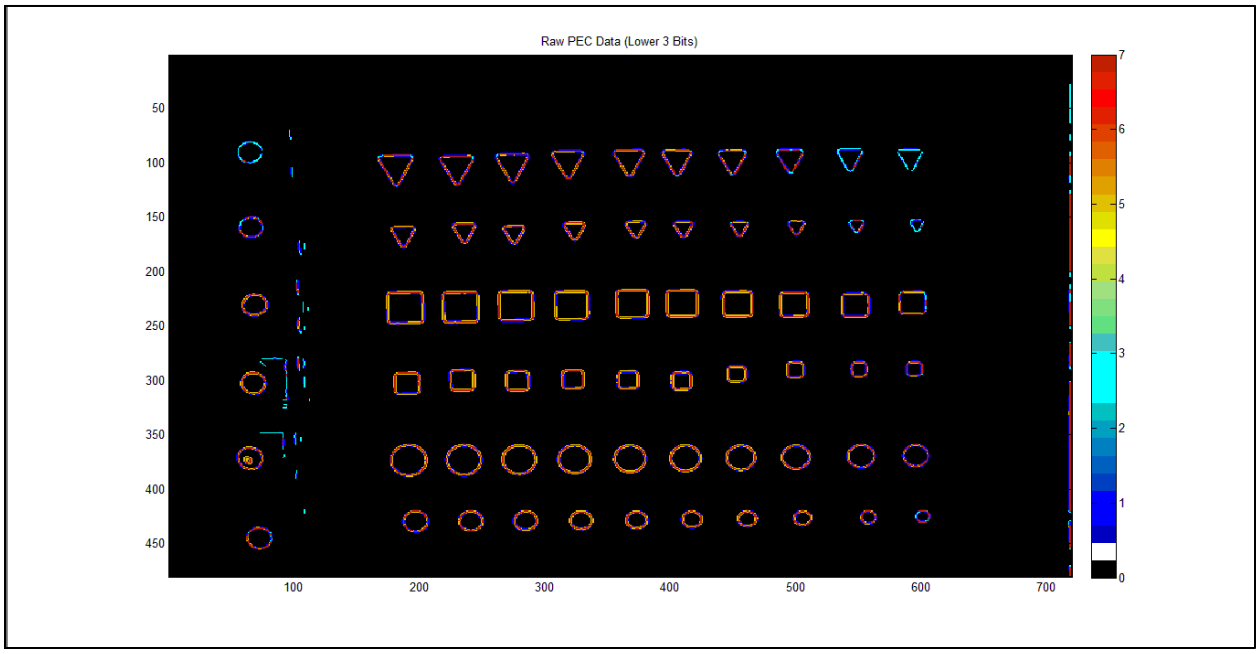


Figure 42. Raw PEC Data (Lower 3 Bits—Magnitude Only)

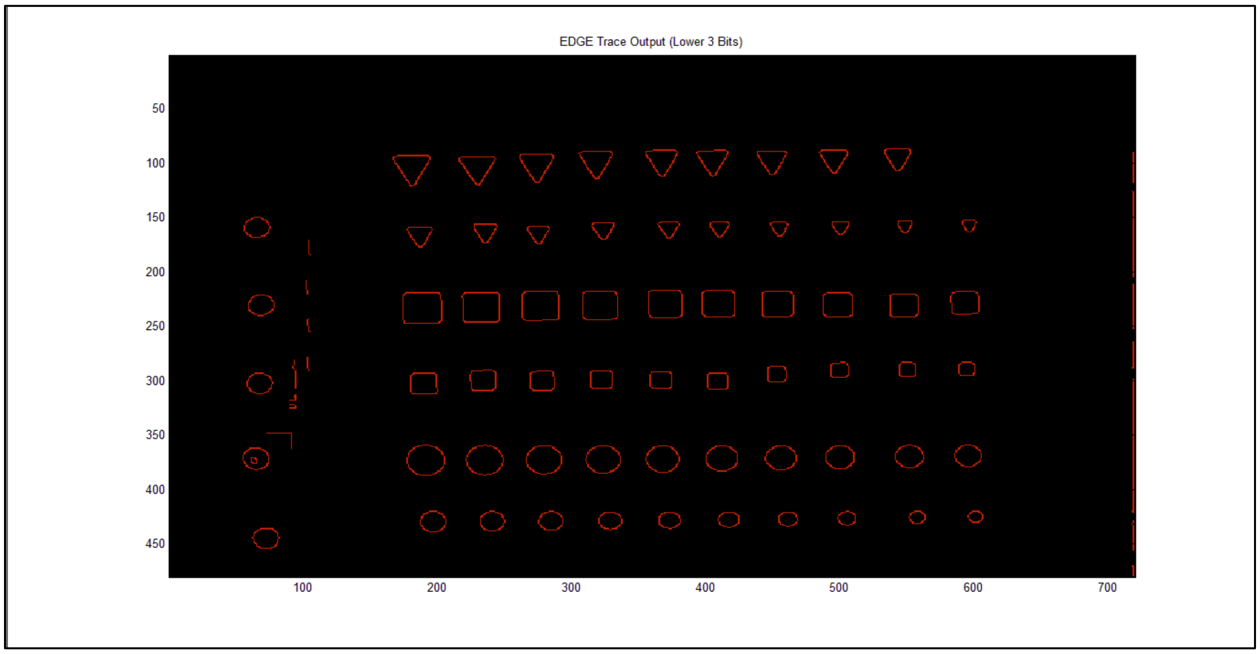


Figure 43. Edge Trace Data (Lower 3 Bits—Magnitude Only)

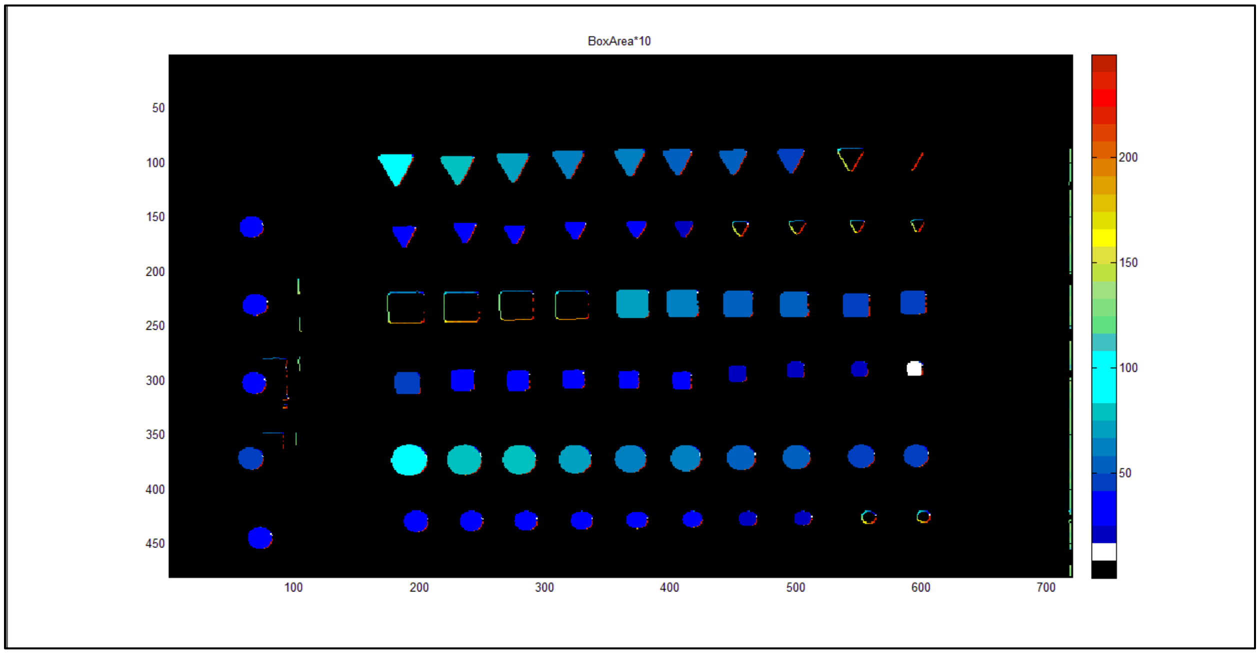


Figure 44. nBoxArea

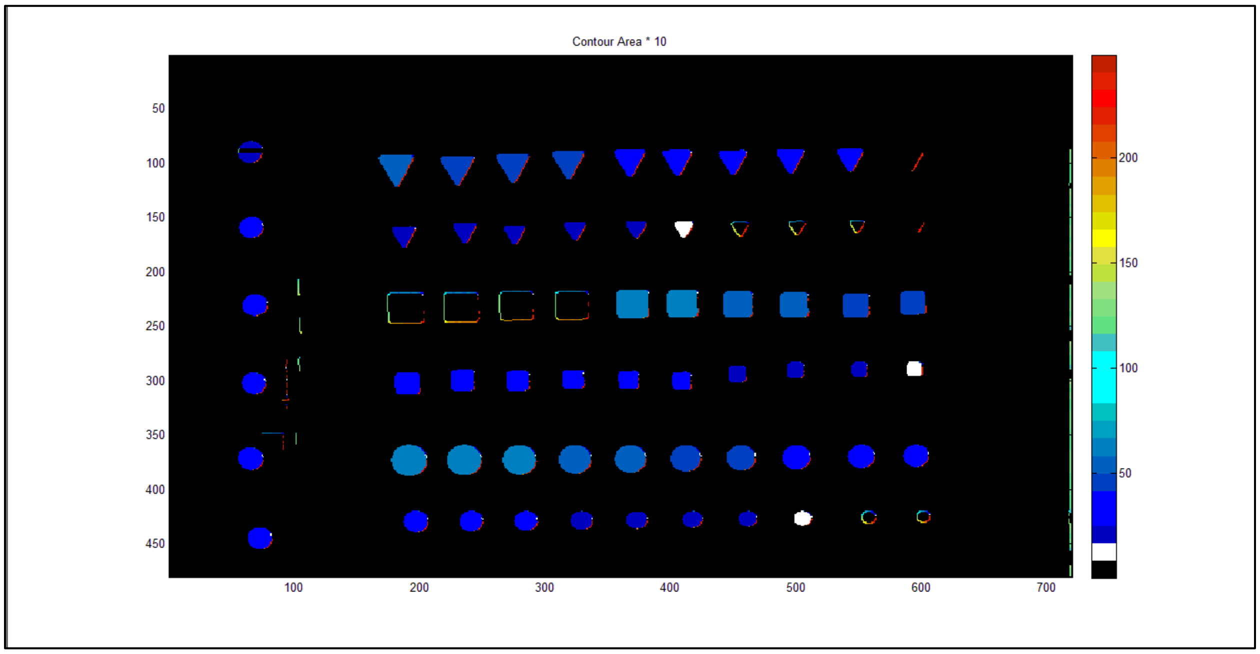


Figure 45. nArea

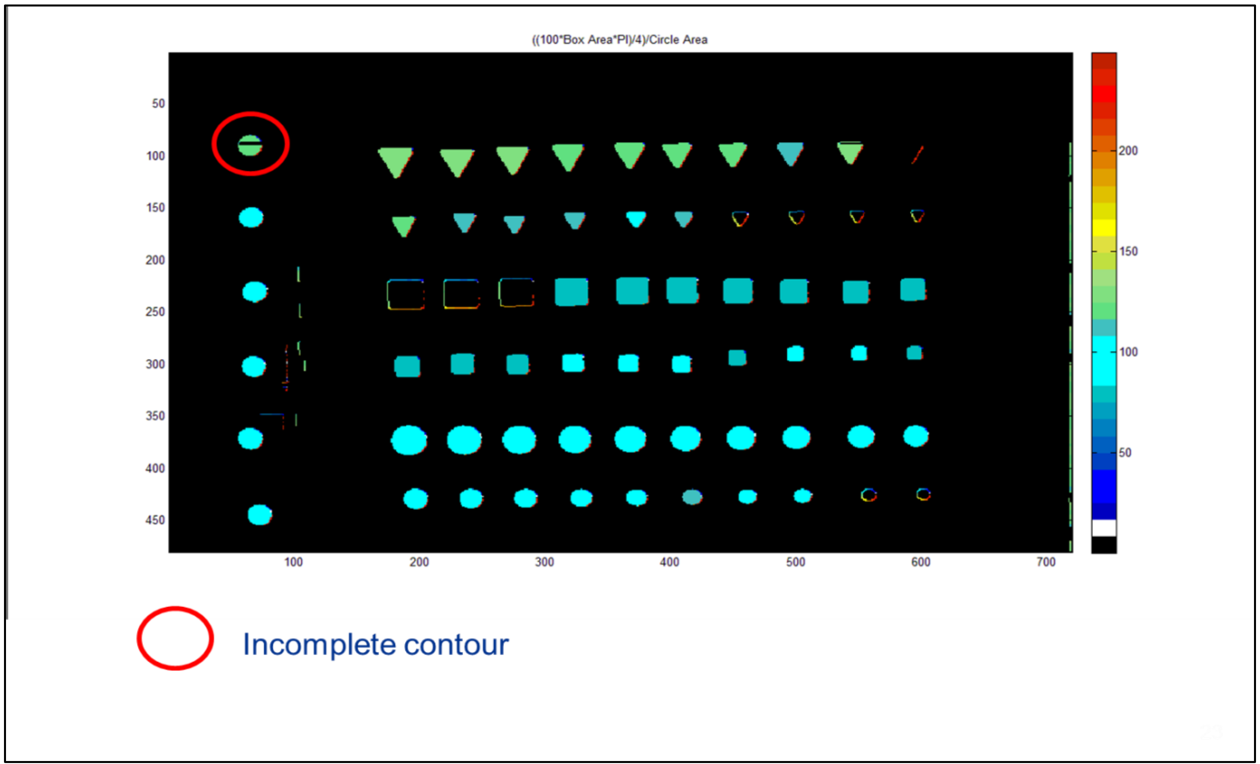


Figure 46. nRatio

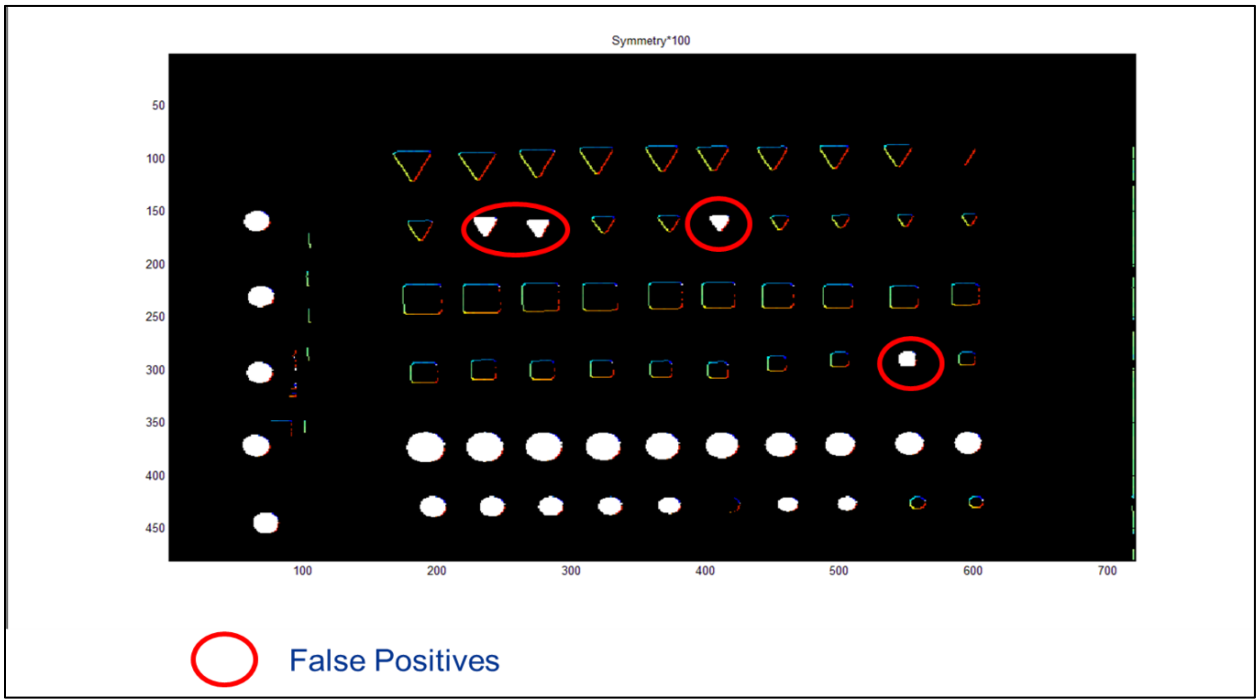


Figure 47. nSymmetry

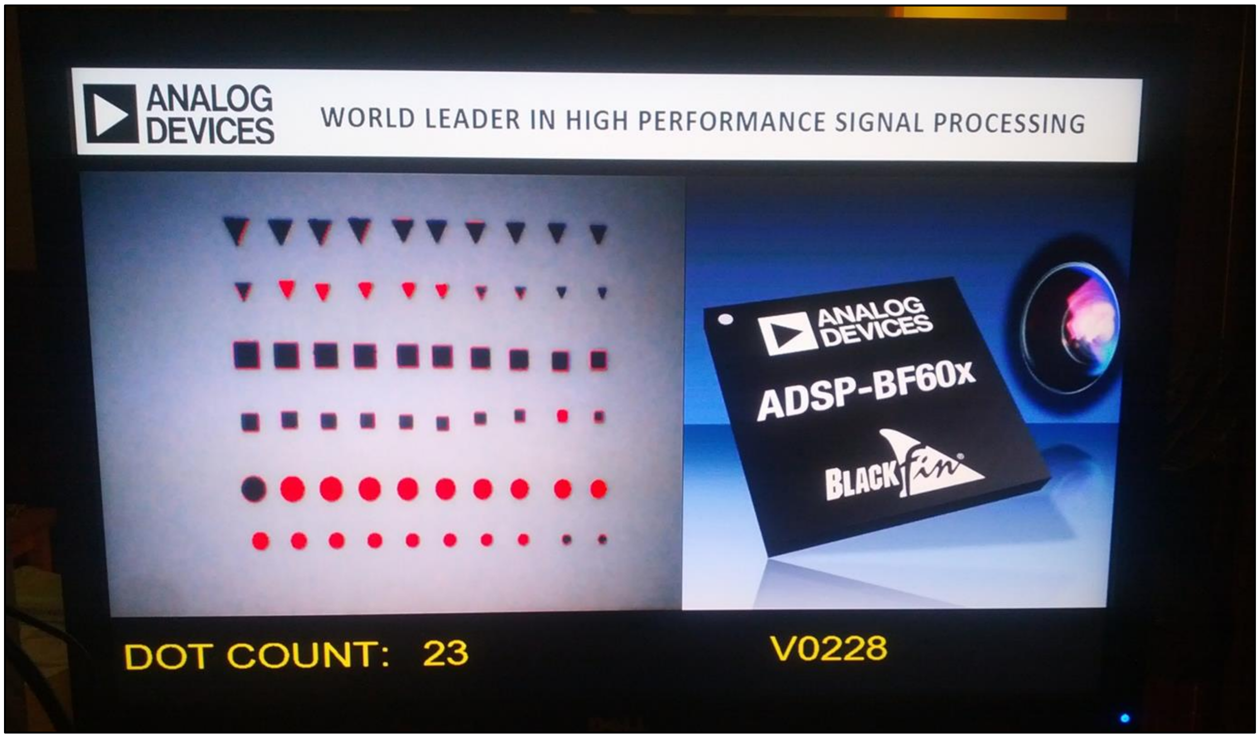


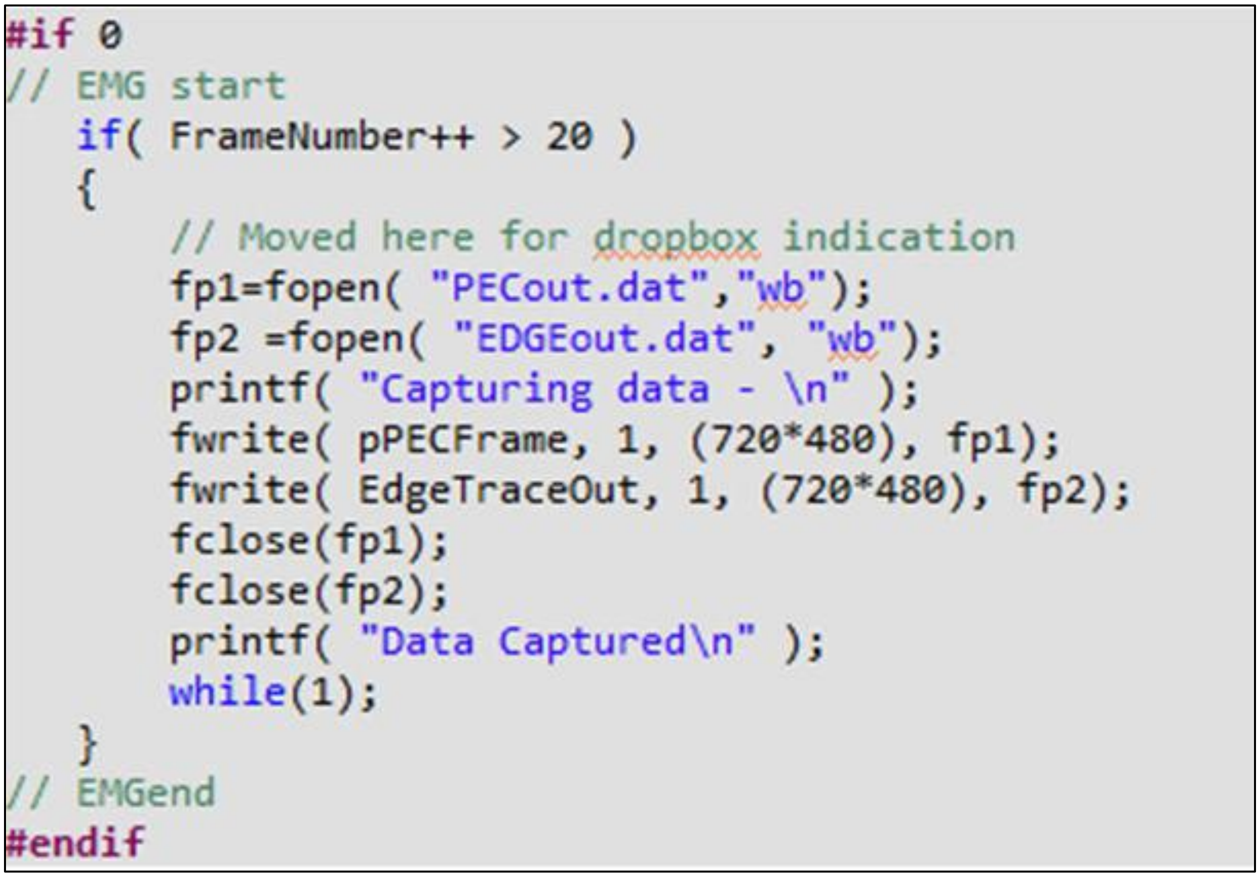
Figure 48. Dots Displayed in Graphics Debug Mode 2

5. Debugging the Algorithm Using Files

The CCES tools have the ability to transfer data from within the BF609 to a file on the host PC's file system. Using this feature, frames of data can be saved for off-chip analysis. This process was used extensively to develop the BDTI Dice Dot-Counting Demo and Reference Design. Using the captured data requires a tool capable of processing and manipulating raw binary files. MATLAB was used for this project to process the raw data and develop and test the algorithm. The graphics contained in the figures in this section were created using MATLAB.

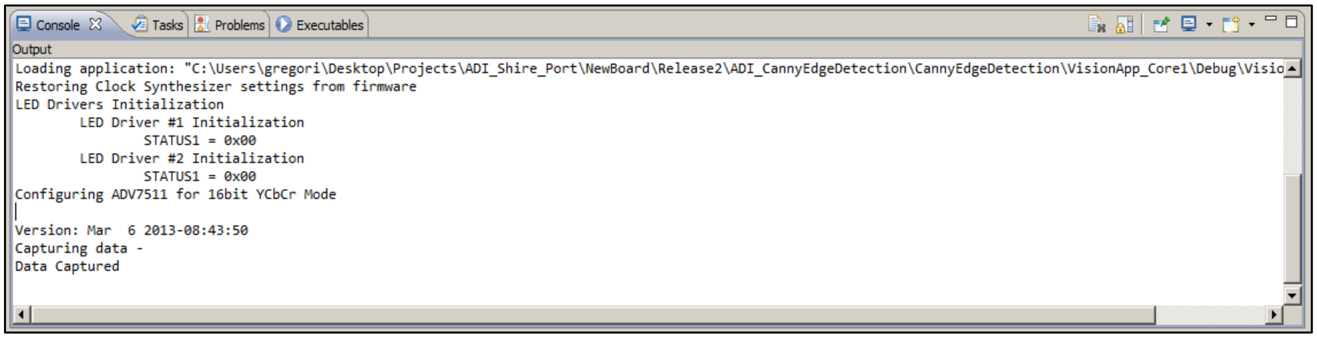
The code to capture the raw data is already in the dot count example. To capture raw PEC or edge trace data, enable the capture code in Core 0. After the CPU captures the raw data, it stalls looping forever, so you can only capture the data from either Core 0 or Core 1, not both at the same time.

Figure 49 shows the code snippet from `EdgeDetection.c` required to catch raw PEC and edge trace data. Simply change the `#if 0` to a `#if 1` to enable raw data capture then build/load the code into CCES. Upon executing the code, the message shown in Figure 50 appears in the console window.



```
#if 0
// EMG start
   if( FrameNumber++ > 20 )
   {
       // Moved here for dropbox indication
       fp1=fopen( "PECout.dat","wb");
       fp2 =fopen( "EDGEout.dat", "wb");
       printf( "Capturing data - \n" );
       fwrite( pPECFrame, 1, (720*480), fp1);
       fwrite( EdgeTraceOut, 1, (720*480), fp2);
       fclose(fp1);
       fclose(fp2);
       printf( "Data Captured\n" );
       while(1);
   }
// EMGend
#endif
```

Figure 49. Capturing Raw PEC and Edge Trace Data



```
Loading application: "C:\Users\gregori\Desktop\Projects\ADI_Shire_Port\NewBoard\Release2\ADI_CannyEdgeDetection\CannyEdgeDetection\VisionApp_Core1\Debug\Visio
Restoring Clock Synthesizer settings from firmware
LED Drivers Initialization
        LED Driver #1 Initialization
                STATUS1 = 0x00
        LED Driver #2 Initialization
                STATUS1 = 0x00
Configuring ADV7511 for 16bit YCbCr Mode

Version: Mar  6 2013-08:43:50
Capturing data -
Data Captured
```

Figure 50. CCES Console Window Showing Capture of PEC and Edge Trace Data

The left side of Figure 51 shows the raw PEC data, while the right shows the edge trace library output. As discussed in Section 4.1.3.3 (Page 16), the PEC outputs a *strong edge*, *weak edge*, or *no edge* classification for each pixel. The PEC edge classification is encoded using 3 bits per pixel. The edge trace output is just 1 bit, all the *weak edges* have either been converted to strong edges or set to 0.

Figure 52 zooms into the raw PEC data. Note the color bar: 7 (or binary 111) is red, 1 (or binary 001) is blue. The PEC has encoded the pixels in the "center" of the edge as strong, while the pixels off the center of the edge are encoded lower. This demonstrates the PEC non-maximum suppression functionality. The result is a thin, well-defined edge after the edge trace library has removed the weak pixels, as shown in Figure 53.

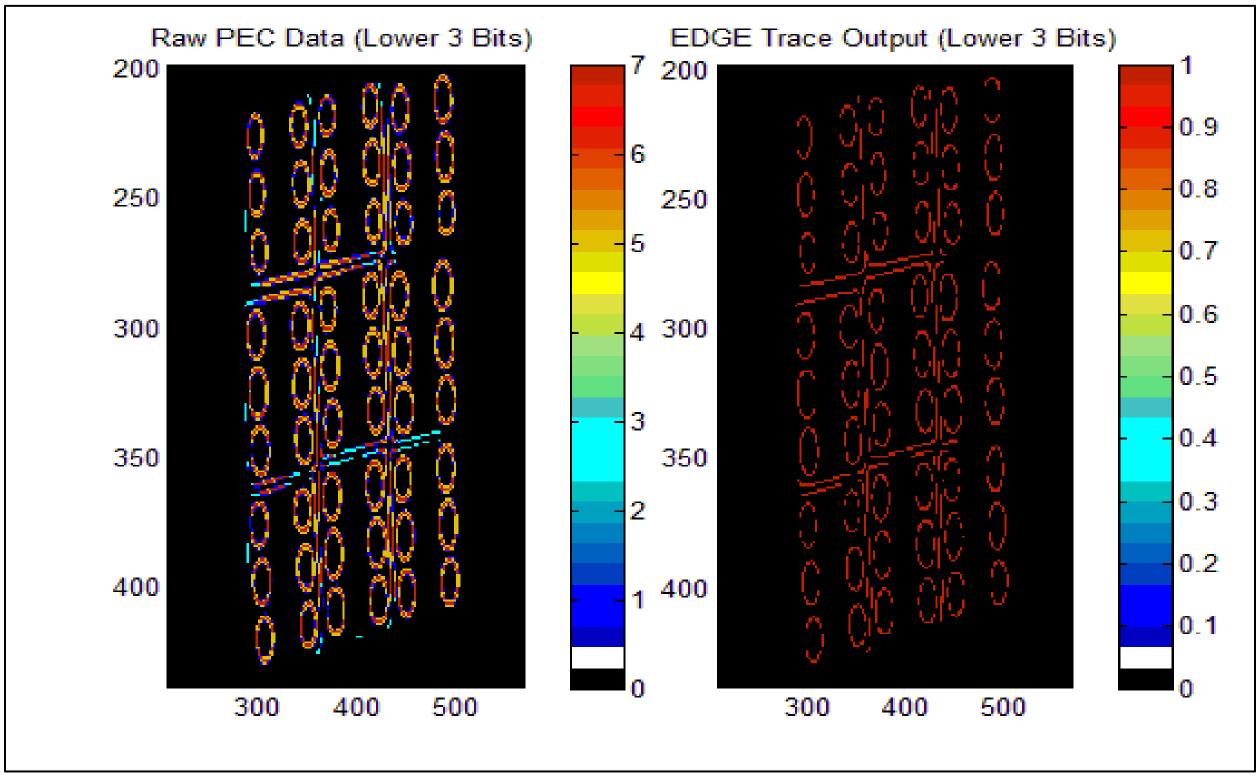


Figure 51. Raw PEC and Edge Trace Data Visualized Using MATLAB

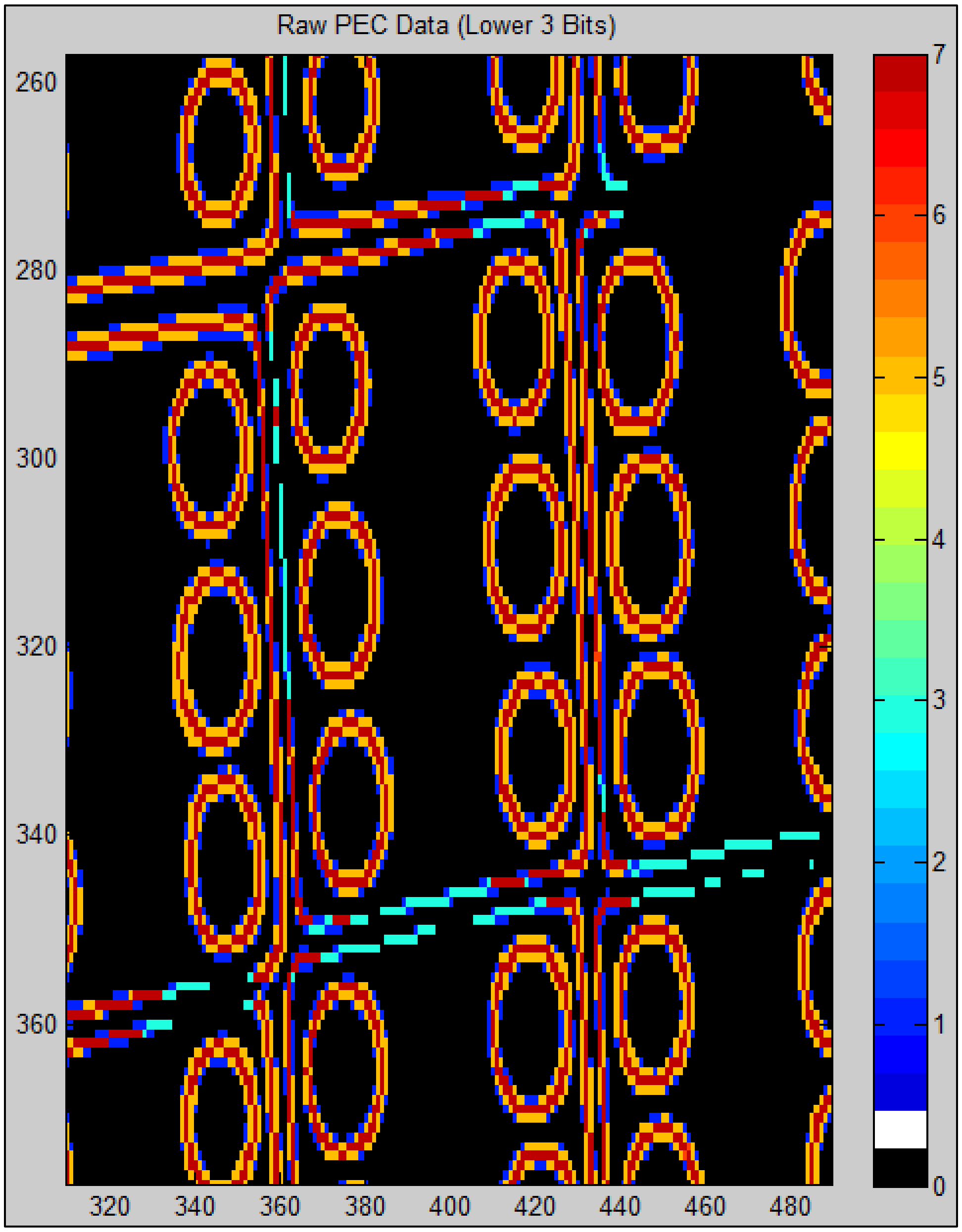


Figure 52. Zooming in on the Raw PEC Data

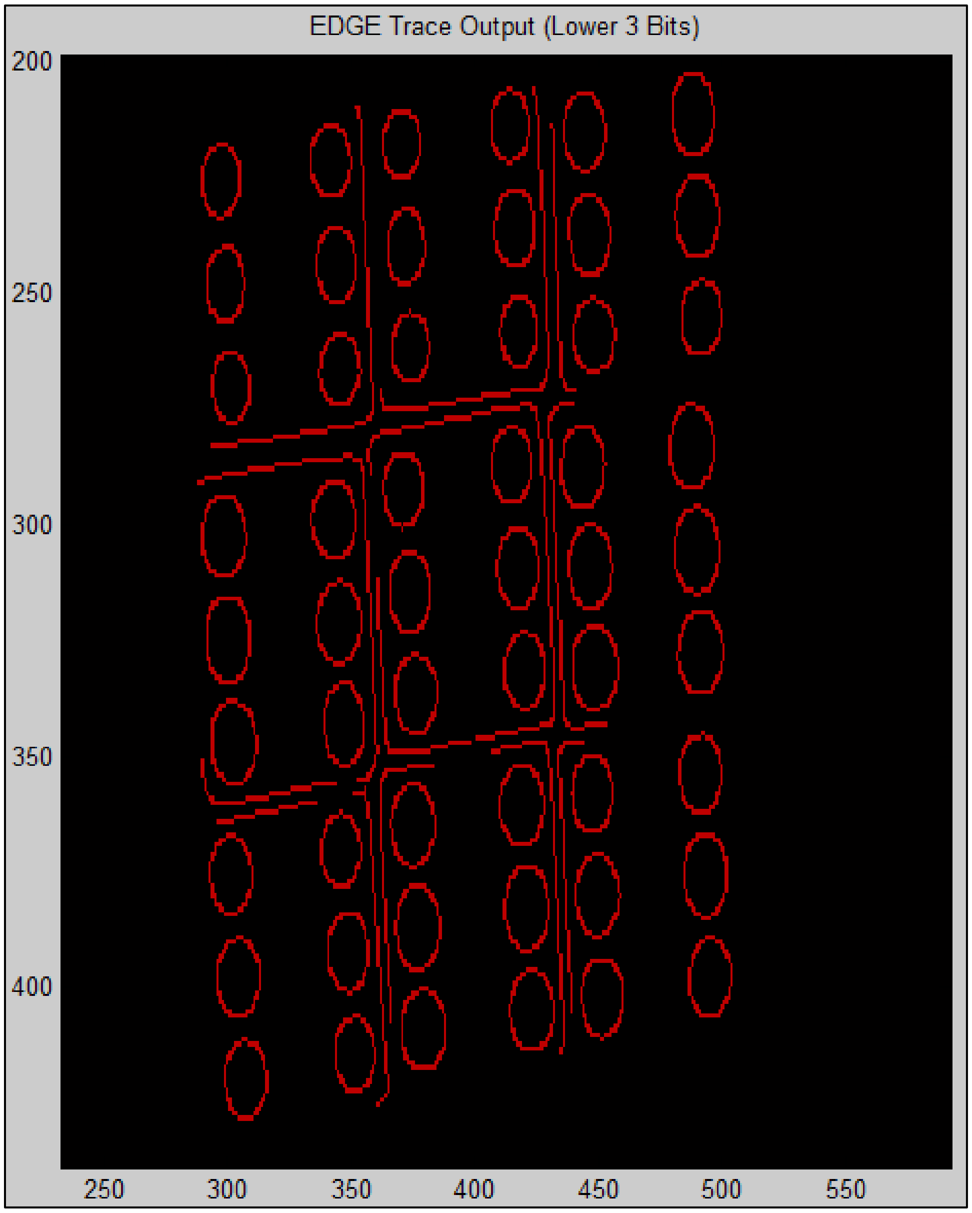
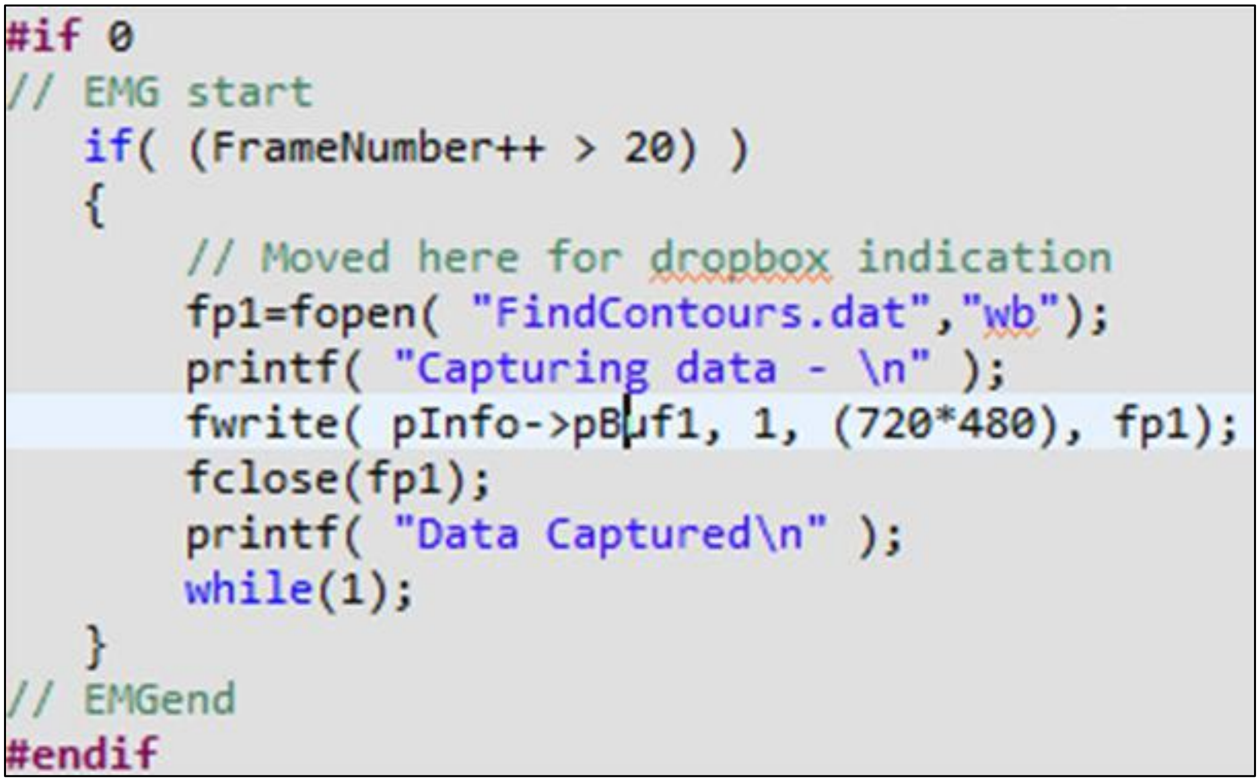


Figure 53. Zoom into the Raw Edge Trace Binary Image

To capture data on Core 1, be sure capture is disabled on Core 0 by replacing the `#if 1` with a `#if 0`. To enable capture on Core 1, find the code shown in Figure 54 in `node_1.c`. Change the `#if 0` to a `#if 1`.

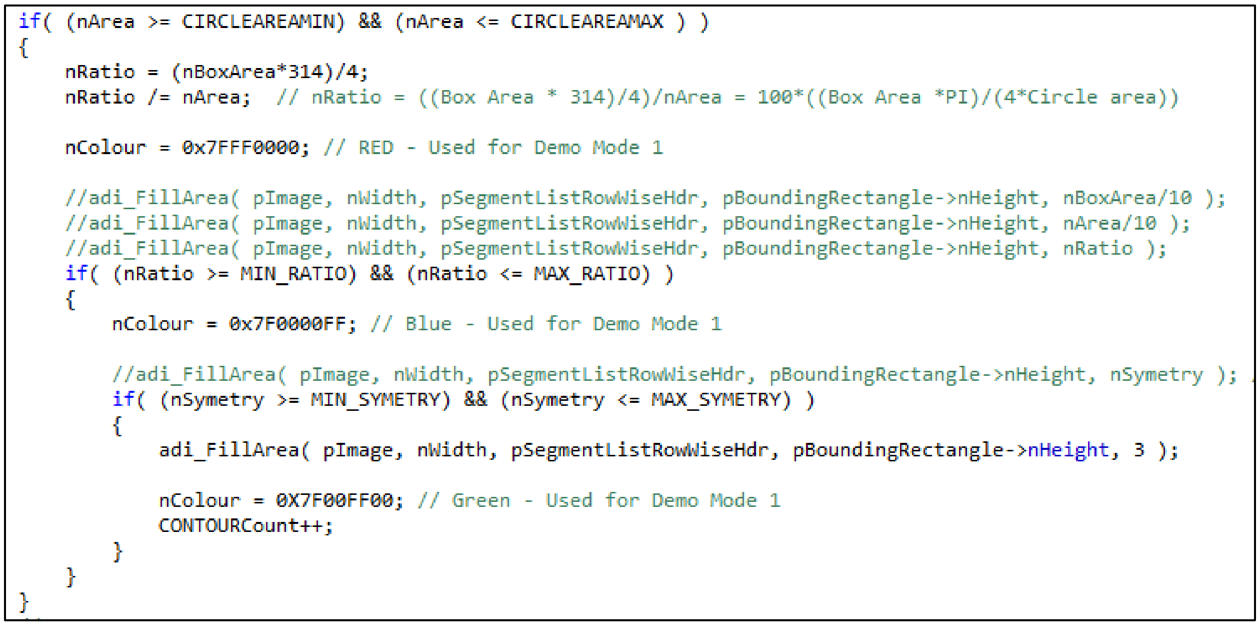


```
#if 0
// EMG start
   if( (FrameNumber++ > 20) )
   {
       // Moved here for dropbox indication
       fp1=fopen( "FindContours.dat","wb");
       printf( "Capturing data - \n" );
       fwrite( pInfo->pBuf1, 1, (720*480), fp1);
       fclose(fp1);
       printf( "Data Captured\n" );
       while(1);
   }
// EMGend
#endif
```

Figure 54. Enabling Raw Capture on Core 1

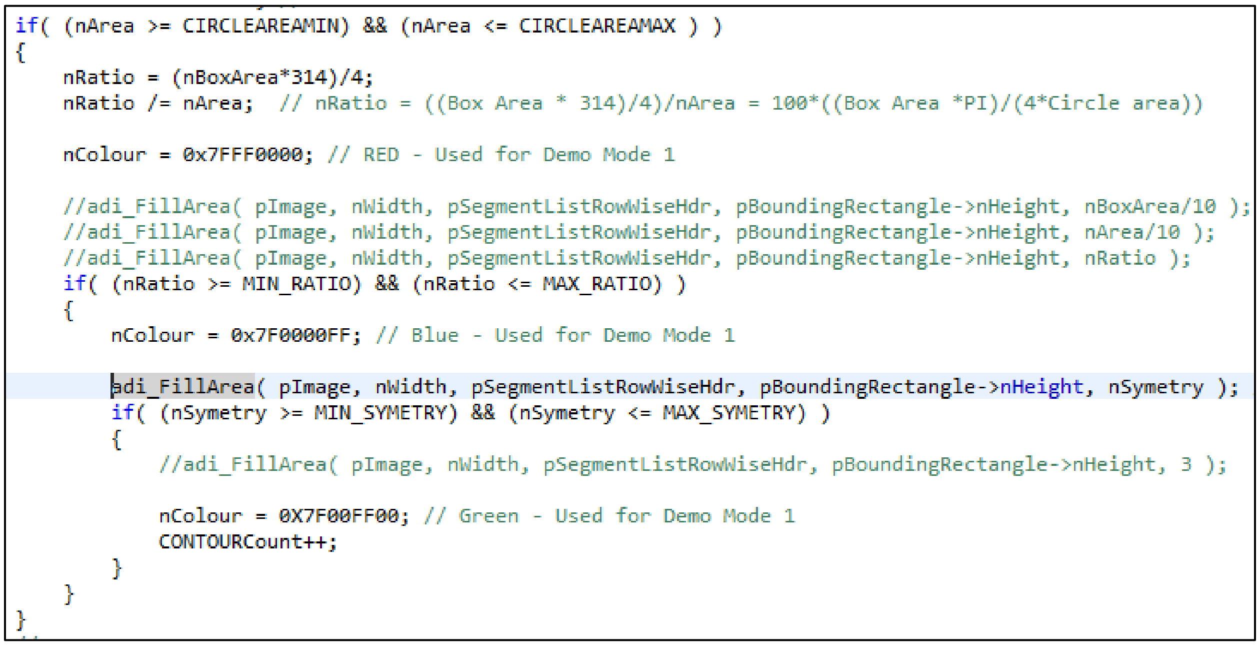
Core 1 runs the `adi_find_contours()` function, which contains the edge classifier. The code is heavily instrumented to provide data on every layer of the classifier. Only one frame can be captured, so the trick is to embed the data in the frame. The `adi_FillArea()` parameters have been modified to include a fill value. Use the `adi_FillArea()` function to fill a region of the frame with test data.

Figure 55 shows the original classifier code. Figure 56 shows a modification used to view symmetry data. This change will fill any contour that gets through to the symmetry layer with the calculated symmetry for the contour.



```
if( (nArea >= CIRCLEAREAMIN) && (nArea <= CIRCLEAREAMAX ) )
{
    nRatio = (nBoxArea*314)/4;
    nRatio /= nArea;  // nRatio = ((Box Area * 314)/4)/nArea = 100*((Box Area *PI)/(4*Circle area))

    nColour = 0x7FFF0000; // RED - Used for Demo Mode 1

    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nBoxArea/10 );
    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nArea/10 );
    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nRatio );
    if( (nRatio >= MIN_RATIO) && (nRatio <= MAX_RATIO) )
    {
        nColour = 0x7F0000FF; // Blue - Used for Demo Mode 1

        //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nSymetry );
        if( (nSymetry >= MIN_SYMETRY) && (nSymetry <= MAX_SYMETRY) )
        {
            adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, 3 );

            nColour = 0X7F00FF00; // Green - Used for Demo Mode 1
            CONTOURCount++;
        }
    }
}
```

Figure 55. Use adi_FillArea() to See Data Within the Classifier



```
if( (nArea >= CIRCLEAREAMIN) && (nArea <= CIRCLEAREAMAX ) )
{
    nRatio = (nBoxArea*314)/4;
    nRatio /= nArea;  // nRatio = ((Box Area * 314)/4)/nArea = 100*((Box Area *PI)/(4*Circle area))

    nColour = 0x7FFF0000; // RED - Used for Demo Mode 1

    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nBoxArea/10 );
    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nArea/10 );
    //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nRatio );
    if( (nRatio >= MIN_RATIO) && (nRatio <= MAX_RATIO) )
    {
        nColour = 0x7F0000FF; // Blue - Used for Demo Mode 1

        adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, nSymetry );
        if( (nSymetry >= MIN_SYMETRY) && (nSymetry <= MAX_SYMETRY) )
        {
            //adi_FillArea( pImage, nWidth, pSegmentListRowWiseHdr, pBoundingRectangle->nHeight, 3 );

            nColour = 0X7F00FF00; // Green - Used for Demo Mode 1
            CONTOURCount++;
        }
    }
}
```

Figure 56. Altering the Code to View Symmetry Data

Figure 57 shows the symmetry data visualized using MATLAB. As explained in Section 3.3.3 (Page 12), the symmetry data is calculated using integer math resulting in a symmetry of 1.0, represented by a value of 10. The minimum symmetry for a dot to pass the filter is 8, the maximum is 11. As shown in Figure 57, only two dots are at the maximum of 11, with a few at the lower level of 9. Most of the dots have a symmetry of 10. Note that the dice edge pixels are not showing symmetry values.

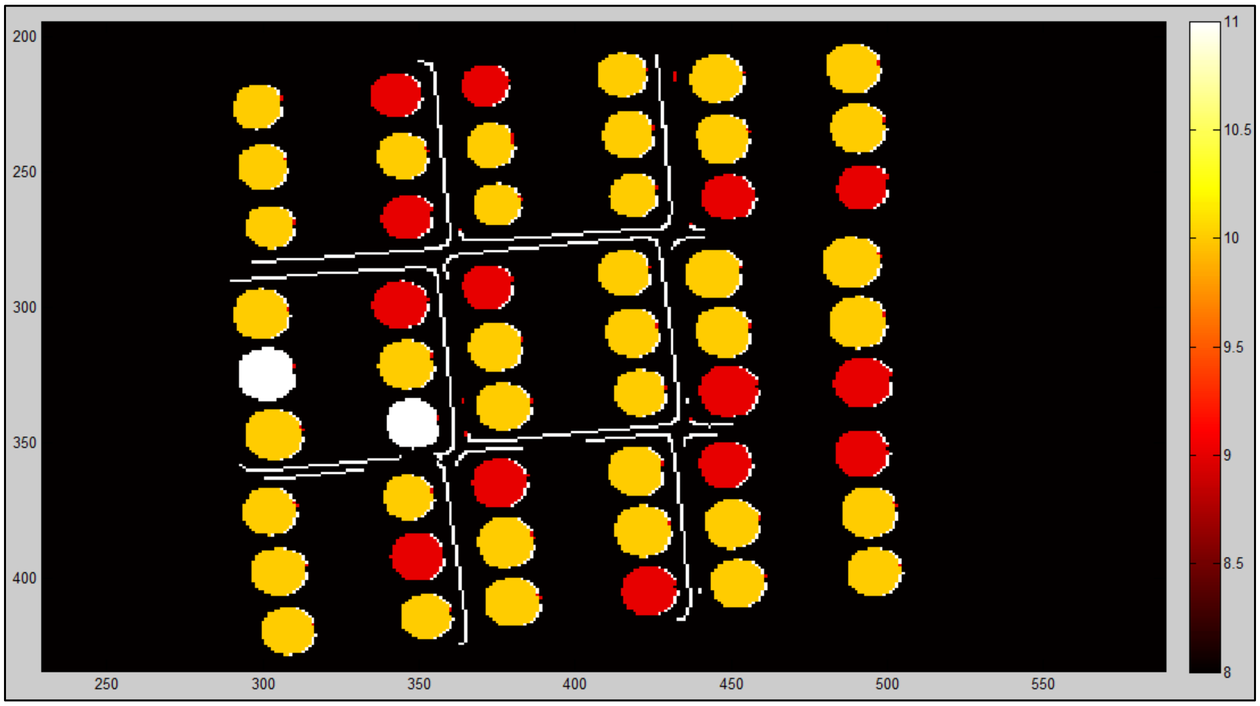
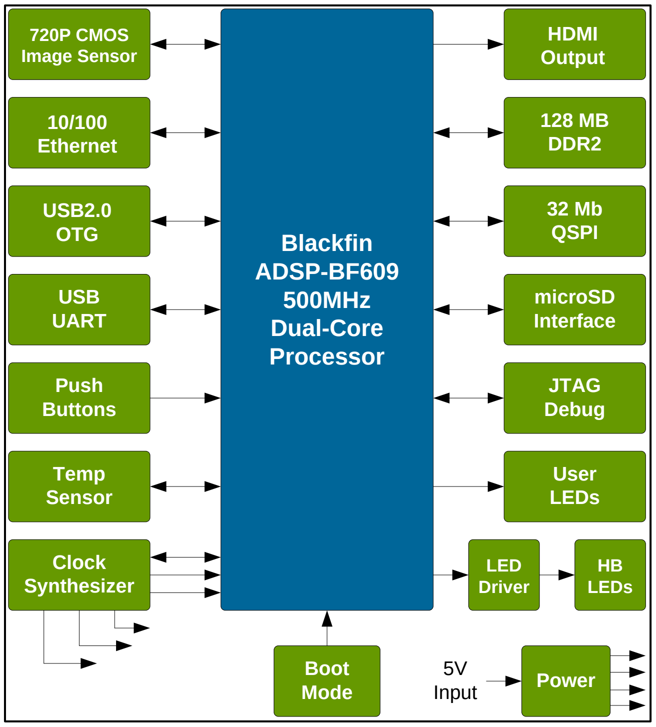
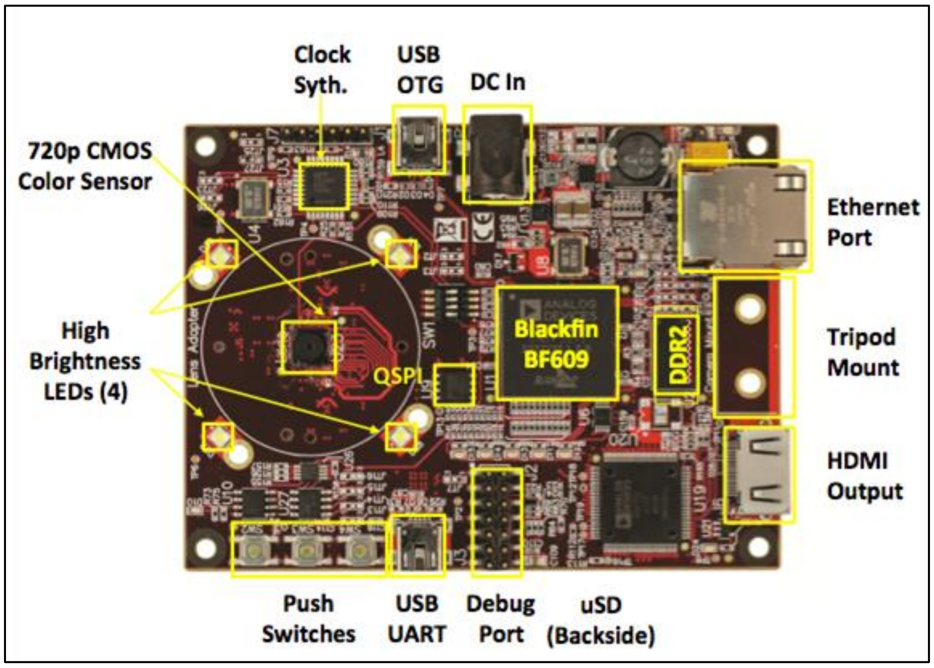


Figure 57. Symmetry Data Visualized Using MATLAB

6. FinBoard: Blackfin® Embedded Vision Starter Kit





The FinBoard is a low-cost board containing an Analog Devices BF609, a 720P color CMOS image sensor, HDMI output, high-brightness LEDs for illumination, Ethernet, USB, and SD interfaces, and 128 Mbytes of DDR2. The board ships with an Edge Detection Framework and the BDTI Dice Dot-Counting Demo and Reference Design, making it easy to quickly get up and running with your own algorithms. You can learn more about the FinBoard at www.FinBoard.org.

7. Conclusion

The BDTI Dice Dot-Counting Demo and Reference Design demonstrates boundary segmentation and classification on the Blackfin ADSP-BF609 dual-core processor using the integrated Pipelined Vision Processor (PVP). The dice dot-counting algorithm is a boundary region classification algorithm testing the boundary region of a contour against a mathematical model. This technique can be applied to shapes other than circles.

Edge detection is the first step for this form of boundary classification. The Canny edge detector is a high performing edge detector usually requiring substantial CPU cycles to complete. The PVP in the BF609 can accelerate Canny edge detection, significantly reducing the load on the CPU/DSP. In addition to the PVP, the BF609 provides dual 500MHz DSP/CPU cores.

The Pipelined Vision Processor (PVP) is a hardware, multi-block, configurable computer vision accelerator. Each block accelerates a computer vision intrinsic. There are 12 blocks total: four to accelerate convolution up to 5×5, two for accelerating Integral Image calculations, and five for accelerating mathematical operations. These blocks are combined using the PVP API provided by Analog Devices. The PVP was configured for Canny edge detection in this example using three convolution blocks and two mathematic blocks.

The BDTI Dice Dot-Counting Demo and Reference Design is built on top of the ADI Canny Edge Detection Framework. The framework provides an easy-to-modify platform for many boundary-based computer vision algorithms.

8. References

[1] Mark S. Nixon & Alberto S. Aguado. Feature Extraction & Image Processing for Computer Vision, Academic Press, 2012

[2] J.R. Parker. Algorithms for Image Processing and Computer Vision Second Edition, Wiley, 2011

[3] http://www.embedded-vision.com/platinum-members/bdti/embedded-vision-training/documents/pages/building-machines-see-finding-edges-i

[4] Gary Bradski & Adrian Kaehler. Learning OpenCV Computer Vision with the OpenCV Library, O'Reilly, 2008

[5] http://www.analog.com/en/processors-dsp/blackfin/adsp-bf609/products/product.html

[6] Product Reference Guide available in the document folder after installing VAT www.analog.com/BF-VAT.

[7] http://www.analog.com/static/imported-files/data_sheets/ADSP-BF6xx_ds_PrF.pdf

[8] E.R. Davies. Machine Vision Theory Algorithms Practicalities, Morgan Kauffmann publications, 2005

[9] http://www.finboard.org