

Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

1-1990

Automated boundary detection of echocardiograms

Rolando Raqueño

Follow this and additional works at: <https://repository.rit.edu/theses>

Recommended Citation

Raqueño, Rolando, "Automated boundary detection of echocardiograms" (1990). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Automated Boundary Detection of Echocardiograms

by

Rolando Raqueño

**A thesis submitted in partial fulfillment of the requirements for
the degree of Master of Science in Computer Science at the
Rochester Institute of Technology.**

December 1990

Signature of the Author **Rolando Raqueño**

Accepted by **Peter G. Anderson**
Coordinator, M.S. Degree Program

**Computer Science
Rochester Institute of Technology
Rochester, New York**

CERTIFICATE OF APPROVAL

M.S. DEGREE THESIS

**The M.S. Degree Thesis of Rolando Raqueño
has been examined and approved by the thesis
committee as satisfactory for the thesis requirement
for the Master of Science Degree**

John Schott

Dr. John Schott, Thesis advisor

Peter G. Anderson

Dr. Peter Anderson

Robert T. Gayvert

Robert Gayvert

19 Dec 90

Date

**Computer Science
Rochester Institute of Technology
Rochester, New York**

THESIS RELEASE PERMISSION FORM

**Title of Thesis : Automated Boundary Detection of
Echocardiograms**

I, Rolando Raqueño, request that the Wallace Memorial Library of the Rochester Institute of Technology notify me prior to reproducing this thesis in whole or in part. I can be reached at the Digital Imaging and Remote Sensing Laboratory of the Center for Imaging Science at the Rochester Institute of Technology.

Rolando Raqueño

Rolando Raqueño

12/20/90

Date

Automated Boundary Detection of Echocardiograms

by

Rolando Raqueño

Abstract:

Two-dimensional echocardiograms of the left ventricle of the heart have been used to produce three-dimensional wire frame reconstructions of the inner wall of the left ventricle chamber. Currently, this process is accomplished by tracing the boundary of the images manually after applying simple image processing algorithms. This manual interaction is a very tedious and time consuming step in the three-dimensional and four-dimensional reconstruction process. It would be desirable to remove as much operator interaction as possible from the process to improve repeatability and decrease throughput time.

This thesis investigated and implemented several basic and novel image processing algorithms which automatically preprocessed the echo images and extracted the boundary information necessary for a reconstruction process. Among the algorithms investigated are common spatial kernel operators such as the Gaussian Convolution Kernel and the Standard Deviation Kernel. Also implemented and discussed are the Robust Automatic Threshold Selector (RATS), a Contour Following Algorithm, and a variation of a QUAD-TREE/Pyramid Resolution data structure. Morphological operations of erosion and dilation were applied to give the closing effect necessary in handling boundary dropouts found in some echo images. An effort has been made to utilize the Gould/Deanza image processing system to execute the algorithms in order to take advantage of its unique architecture.

The accuracy of the final sequence of algorithms was tested on actual images and a comparison made between hand drawn borders and the computer generated borders traced on actual echo images.

Keywords: Pattern Recognition, Medical Imaging, Image Processing, Echocardiograms, Boundary Detection

Acknowledgements

I would like to thank everyone at the Center for Imaging Science especially the staff and students (both past and present) of the Digital Imaging and Remote Sensing Lab for both their technical and moral support.

Special thanks to Steve Schultz for his help and invaluable expertise with the Gould/Deanza image processing system, Dr. Carl Salvaggio and Dr. Roger Easton for their editorial help and input, Dr. Schott for giving me the opportunity to work and play on this and several other projects (John, this is all your fault...), and Miss Nina Gibson for being my *de facto* managing supervisor and contract officer/technical reviewer. Thank you everyone.

It truly has been a labor of love. ♥

Dedication

To Mom, Dad, Judith, Junior, and Nina.
Thank you for all your love, patience, and support.
You guys are what life is all about.

And for all the trees whose sacrifice made
this thesis possible ...

Recyclable and made from recycled paper



Table of Contents

List of Tables

List of Figures

1. Introduction	1
1.1 Two-dimensional Sonography Overview	1
1.2 Three-dimensional Reconstruction Overview	4
1.3 Computer Software/Hardware Overview	10
2. Image Processing Hardware	11
3. Image Processing Software	15
4. Algorithms	16
4.1 Sector Region of Interest	17
4.2 Spatial Convolution	18
4.3 Robust Automatic Thresholding Selection	26
4.4 Quad-Tree/Multiresolution Data Structure	39
4.5 Contour Following Algorithm	45
4.6 Morphological Operations	49
4.7 Algorithm Integration	58
5. Results	60
6. Conclusion	72
7. Recommendation	73
References	77
Appendix A: Result Table Summary	
Appendix B: Main Program Listings of Algorithms	
Appendix C: Subprogram Listings of Algorithms	
Appendix D: Image Processing Interface Library Help File	
Appendix E: Block Diagram of DVP	

List of Tables

Table 1 - Threshold Statistics	62
Table 2 - Morphological Processing Statistics.	65
Table 3 - Trace Statistics	70

List of Figures

Figure 1 - Transducer position and corresponding image.	3
Figure 2 - Hardware used in processing echo images.	6
Figure 3 - Four-dimensional reconstruction steps.	8
Figure 4 - Overall image processing system.	12
Figure 5 - Workstation hardware configuration.	14
Figure 6 - Sequence and flow of algorithms.	16
Figure 7 - Sector mask operation.	18
Figure 8 - Smoothing Convolution.	18
Figure 9 - Varying spatial frequency in the horizontal direction.	19
Figure 10 - Serial computation example of correlation (or convolution).	22
Figure 11 - Parallel computation example of correlation (or convolution).	24
Figure 12 - 15x15 Gaussian convolution kernel.	25
Figure 13 - Histogram based grey level threshold selection.	26
Figure 14 - Edge scan line (grey).	28
Figure 15 - Edge scan line derivative (grad).	29
Figure 16 - Derivative kernels.	32
Figure 17 - Grey-grad values.	33
Figure 18 - Flow of RATS algorithm.	37
Figure 19 - Traditional Quad-Tree structure diagram.	40
Figure 20 - Modified multiresolution representation.	44
Figure 21 - Contour following kernel.	45
Figure 22 - Contour following.	47
Figure 23 - Closed contour cases and corresponding area contribution convention of contours and vectors.	48
Figure 24 - Example of morphological operation (dilation).	51
Figure 25 - Sample morphological closing sequence.	53
Figure 26 - DVP implementation of morphological operations.	55
Figure 27 - Quad-Tree type subdivision search.	57
Figure 28 - Original images (masked).	60
Figure 29 - Smoothed images.	61
Figure 30 - Thresholded Images.	62
Figure 31 - Threshold plot.	63
Figure 32 - Multiresolution images.	64
Figure 33 - Morphologically processed images with traces.	65
Figure 34 - Computer and Manual Traces on Original Images.	66

Figure 35a - D180F00 trace plots	67
Figure 35b - D180F03 trace plots	67
Figure 35c - D180F09 trace plots	68
Figure 35d - D180F12 trace plots	68
Figure 35e - D180F18 trace plots	69
Figure 35f - D180F22 trace plots	69
Figure 36 - Computer vs. Manual Trace Area	70
Figure 37 - Ventricular Area vs. Image Frame Number	71
Figure 38 - Scaled sector outlines	74
Figure 39 - Trace plots of scaled sector traces.	74
Figure 40 - Trace plots of normalized sector traces.	75
Figure 41 - Trace plots of rotated sector traces.	75

1. Introduction:

One of the most popular medical imaging modalities used today is the echocardiogram (sonogram of the heart area). This diagnostic technique utilizes ultrasonic signals to image various sections of the thoracic anatomy. At present, echocardiograms provide a real-time tomographic sampling of the heart walls and associated chambers. The current trend of echocardiograms, as in other medical imaging modalities, is the extension of the technology into three and four dimensions. The objective of this thesis is to introduce and evaluate several image processing algorithms used for automated boundary detection -- a necessary step for feature extraction in the extension of two-dimensional information representation to spatial and temporal dimensions.

An overview of the basic principles of sonography will be discussed to present its advantages and limitations and a survey of research work in echocardiogram reconstruction will be enumerated. The details of a specific three-dimensional reconstruction regimen will be presented to convey the necessity for the key step of automatic boundary detection. Lastly, an overview of the hardware, software, algorithms, results, and areas of further investigation will be presented.

1.1 Two-Dimensional Sonography Overview:

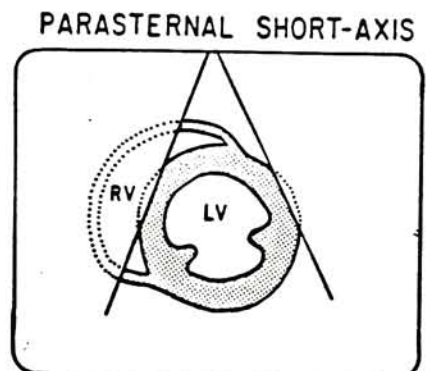
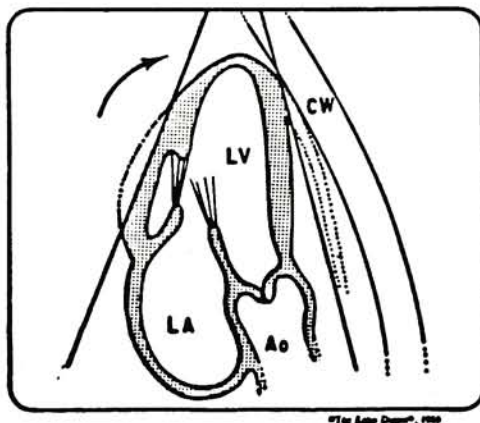
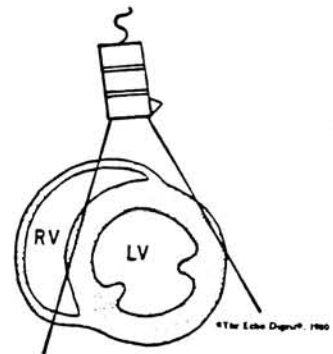
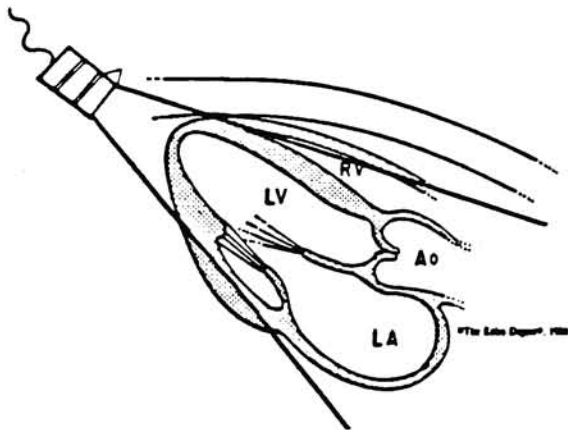
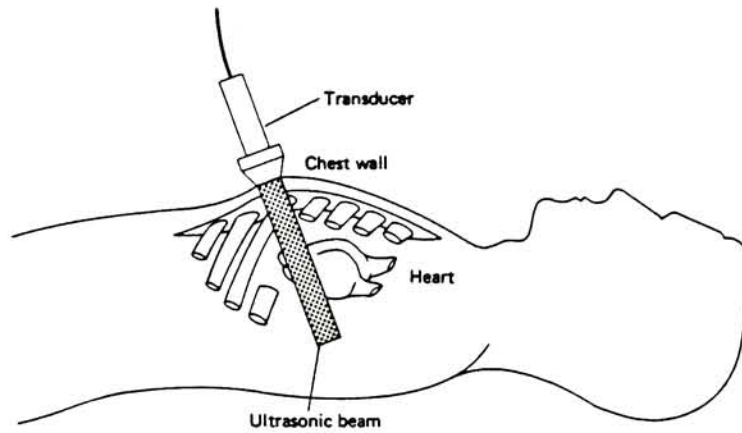
Two-dimensional sonography is a diagnostic medical imaging technique which uses ultrasonic energy to form a two-dimensional tomographic image of an anatomical feature. Sonography operates on the principle of dissimilar acoustic impedance of tissue interfaces. Propagation of ultrasonic signals through each tissue produces a relative signature based on the tissue type and the surroundings. This information coupled with the physician's knowledge of the anatomy can yield a great deal about the structural condition of an organ (Feigenbaum 1981).

The actual ultrasound hardware utilizes an array of piezoelectric crystals to act as a transducer/receiver of the ultrasonic signal. These piezoelectric crystal transducer elements undergo bulk distortion when subjected to an electric signal and conversely produce an electric signal when distorted by some force. As a transducer, the crystals propagate 3 to 5 megaHertz ultrasonic pulses into the body where the signals are attenuated, reflected, and refracted by the different tissue types and their interfaces with other tissues or body cavities. The different types of tissues in the body and surrounding structures each have a characteristic reflective, absorptive, and transmissive property to the ultrasound signal. The degree of attenuation and return time of the reflected ultrasonic pulses to the transducer indicates the relative densities of the tissues as well as their physical location.

A special form of ultrasonography known as **phased array sector scanning** pulses an array of crystals under computer control and takes the returning signals and maps them into a two-dimensional raster image of the sector slice subtended by the array of transducers. This results in a tomographic view of the scanned anatomy. **Figure 1** gives a diagrammatic representation of the transducer location and the corresponding image as it would be seen on the display screen (Cromwell 1980, Brown 1980).

The appeal of echocardiograms lies in its non-invasive nature and relatively low cost. It does not expose the body to the traumas experienced during conventional exploratory surgery. Unlike other modalities such as x-ray angiography and radionuclide scintigraphs which use radiation and radioactivity to form images, the ultrasonic signals generated by sonogram hardware pose no known health hazard. The cost of ultrasound equipment and its maintenance cost is considerably less than computerized axial tomography (CAT) or magnetic resonance imaging (MRI) systems. Another significant advantage of echocardiograms not common in other imaging modalities is its high sampling rate. This allows imaging and recording of events at video rates -- a necessity for studying the dynamic nature of the heart.

Sonograms do have a major drawback in that the spatial resolution is much lower than other imaging modalities. Sonograms generally resolve features of approximately one millimeter in the smallest dimension. There is also a compromise between resolving power and signal penetration. The resolving power increases as the frequency of the ultrasonic signal increases, but the depth of penetration of the ultrasonic pulse into the body decreases. Sonograms are also characterized by a relatively low signal-to-noise ratio. The main source of noise is attributed to speckle noise due to the tissue inhomogeneities and nonuniform reflection of the signals. Although these low signal-to-noise ratio images are satisfactory for human visual inspection, they pose significant problems in terms of automated image analysis. Nevertheless, the cost effective utility of the echoimaging modality is exemplified by its popularity in the medical diagnostic community. This interest is further supported by the research that has been done in attempts to extend two-dimensional images to three-dimensional data.



THE APICAL LONG AXIS VIEW
 Transducer Orientation to the Chest Wall
 vs. the Video Image Displayed

THE PARASTERNAL SHORT AXIS VIEW
 Transducer Orientation to the Chest Wall
 vs. the Video Image Displayed

Figure 1 - Transducer position and corresponding image.

1.2 Three-Dimensional Reconstruction Overview:

The trend from two-dimensional to three-dimensional representation is a necessary step for quantitative diagnosis. At present, the two-dimensional images give a fragmentary view of the anatomy. Physicians currently have to study several two-dimensional images from different transducer positions and mentally reconstruct a solid representation of the organ being scanned. This becomes more difficult in the case of the heart because of the additional motion information involved. This mental reconstruction exercise requires considerable skill and experience on the part of the physician to correctly merge the images into an overall picture. The resulting mental image is qualitative and temporary. Because of the lack of precise measurements and permanent hardcopy records of what the physician observes, there is great interest in developing representative models to give immediate insights into diagnosis of heart disorders. With this quantitative information, long term predictive assessments can be used as a preventive measure against future ailments.

The majority of previous research has focused mainly on the left ventricle because it is the critical chamber which pumps oxygenated blood to the rest of the body and is affected by many pathologies due to the mechanical stresses from repeated contraction and relaxation. The ability to model the left ventricle and its chamber boundary as it progresses along the cardiac cycle can provide an enormous amount of empirical data relating to the pumping efficiency of the heart and the condition of the heart walls. Quantitative measurements can theoretically allow the modelling of blood flow (hemodynamics) within the left ventricle. Knowledge of the hemodynamics not only gives a gauge on heart pumping efficiency, but also a measure of the cardiac wall condition and corresponding mechanical stresses. This gives a locator as well as a predictor of areas within the left ventricle which may be susceptible to coronary disorders.

Sawada, 1983 provided an early technique of three-dimensional reconstruction from randomly recorded multiple two-dimensional phased array sector scanned echo images. The images were of short axis type giving a circular view of the left ventricle chamber. These were digitized into a computer, but no image processing was applied. The boundary information for these images were manually traced. Likewise, Nixon, 1983, was interested in three-dimensional echoventriculography (volume determination) and Chandra, 1983, was investigating normal diastolic elastic properties of the left ventricle. Both used some form of three-dimensional reconstruction to generate the surface description of the ventricle but without any reference to image processing or automatic boundary detection.

Skorton, 1981 describes an application of digital image processing on echocardiograms with emphasis on identification of the endocardium (inner chamber wall) using frame averaging and thresholding to generate an initial boundary approximation. This process was combined with a Sobel edge operator which produced a boundary confirmation edge map to improve the accuracy of the boundary approximation in the thresholding step. The resulting boundary was

traced manually based on the two processes. Matsumoto, 1981 applied simple image processing techniques of size reduction, smoothing, thresholding, and differentiation. The borders were then traced manually to generate the final reconstruction. Adam, 1987 developed a semiautomated technique for border tracking of cine echocardiograms beginning with a manually traced image. This initial image was used as a guide for border extraction for subsequent images. Accompanying image processing algorithms included a fast median filter, center of gravity calculation, a spatially dependent contrast stretch operator, and a statistical estimator of possible boundary points. McCann, 1987 made significant improvements in the acquisition of images by developing a mechanical device to increase sampling views to achieve a better reconstruction. The resulting reconstruction also attempted to preserve the backscatter to give a raster volume image instead of a wire frame surface description. A semi-automatic operator-interactive algorithm is applied to extract the boundary information.

In each cited case, an operator was necessary at some stage of the reconstruction to manually extract boundary information or guide a set of algorithms to do so. The following data set and reconstruction method by Schott *et al.*, 1987 will be the basis of the automation attempt of this thesis because of the availability of the imagery and supporting software.

Figure 2 diagrams the hardware involved in the processing of the echo images. The images were supplied by the Echocardiography Lab at the University of Alabama School of Medicine at Birmingham in standard VHS video recorder format recorded in real-time. The selected images were re-recorded onto a video-disc recorder and in turn freeze-framed by a time-base corrector and digitized by a Gould/Deanza IP8500 image processing system. The image processing system contains twelve 1024 x 1024 x 8 bit memory planes to store imagery for fast pipeline processing by a Digital Video Processor (DVP) and display on a high resolution color monitor. The images were digitized at a 512 x 512 spatial resolution with a 256 grey-level resolution from a standard NTSC signal and then stored on the host VAX 8350's disc storage.

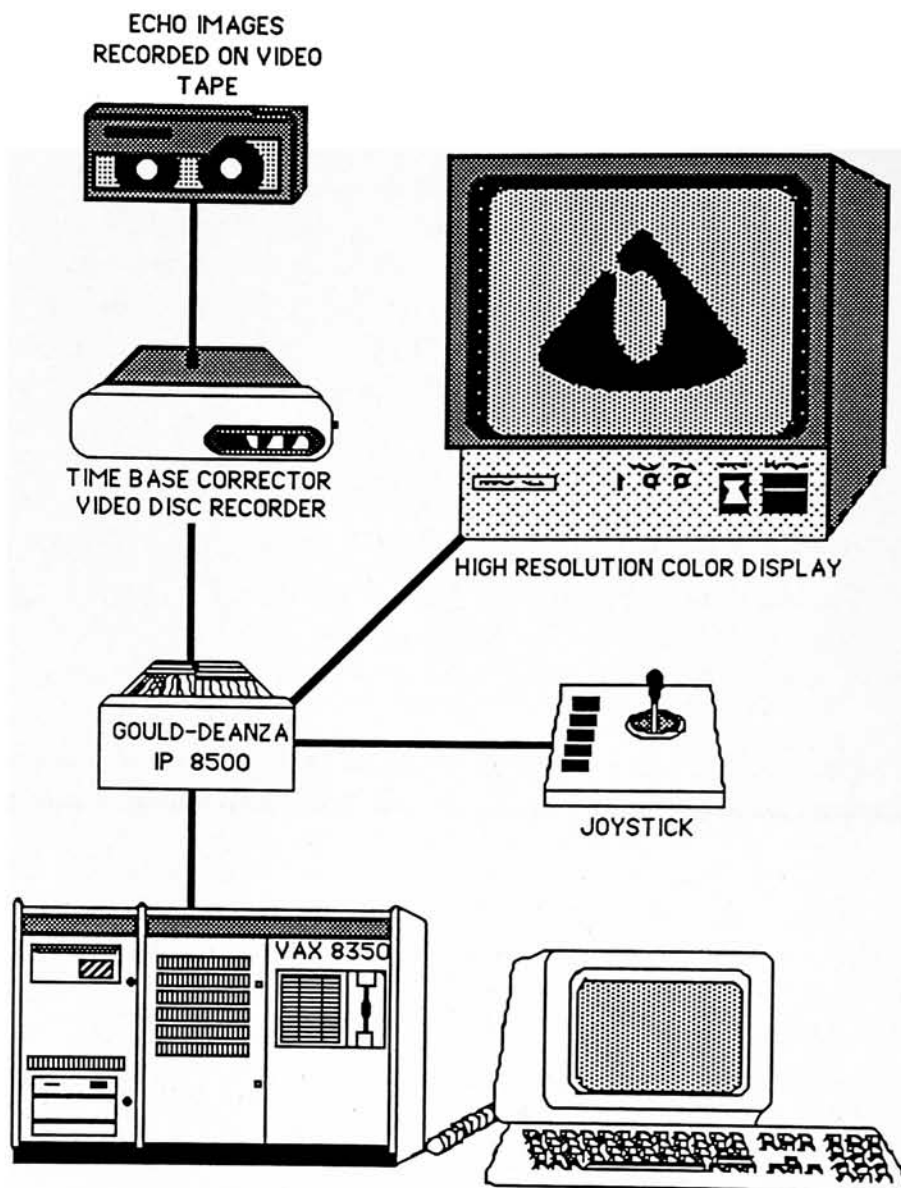


Figure 2 - Hardware used in processing echo images.

Figure 3 illustrates the processing steps of the four-dimensional reconstruction. The recorded images were real-time (video rate) samplings of the left ventricle contracting and relaxing for several cardiac cycles at varying transducer positions. The transducer was apically positioned and incrementally rotated about the major axis of the left ventricle every 30° (through a 180° sweep) creating a cone of revolution subtending the left ventricle from which discrete samplings were made. For each of the transducer positions (tomographic plane), a subset of images representative of a complete cardiac cycle was digitized. Each selected image was then processed by convolving the image with a 3×3 convolution kernel for a smoothing effect and removal of salt-and-pepper noise. Each smoothed image was then radially differentiated from the left ventricle center to visually enhance the endocardium borders. A trained operator then manually traces the borders using a joystick input device.

For each of the tomographic planes, a 9 image sequence cycle of the left ventricle from diastole to diastole were sampled. These samplings of 9 images were necessary to allow interpolation of the heart chamber boundary with respect to time in order to compensate for minute variations of heart rates from cycle to cycle. This boundary-time interpolation allowed a boundary which was not represented by an actual image to be predicted from two other sampled boundaries. Altogether, a total of 54 images were processed and traced by the operator. To achieve more accurate results, a finer increment of transducer rotation of about 10 degrees was required. This increases the total number of images to process to 162 images. Several wire frames as a function of time can now be generated by time interpolations. On a more realistic level, an average cardiac cycle of 60 beats per minute can consist of as many as 30 frames of imagery. This translates to as many as 270 images needed to produce a wire frame representation of the left ventricle contracting through a cardiac cycle.

Processing Procedure

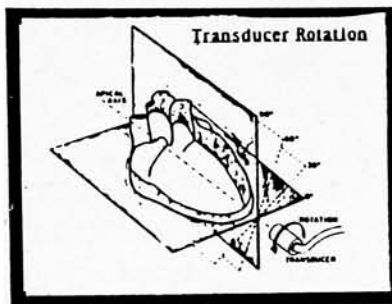


Image Processing and Boundary Trace

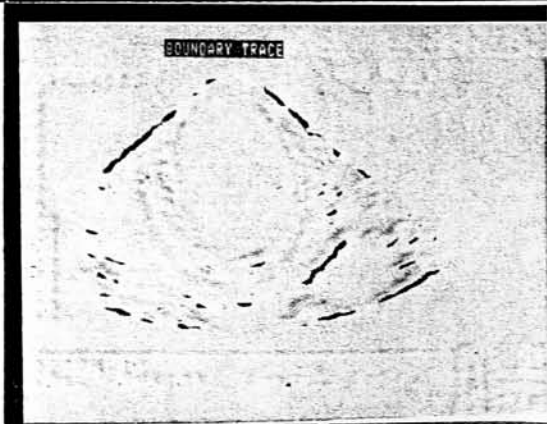
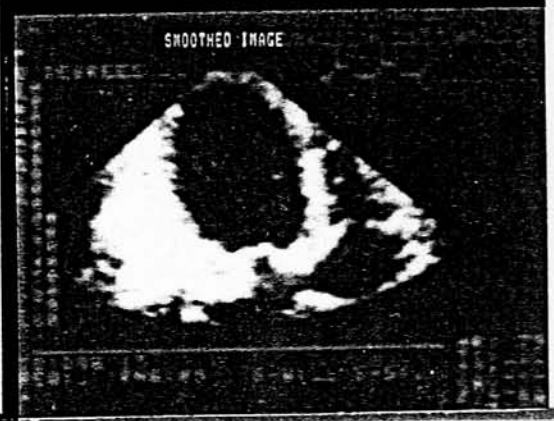
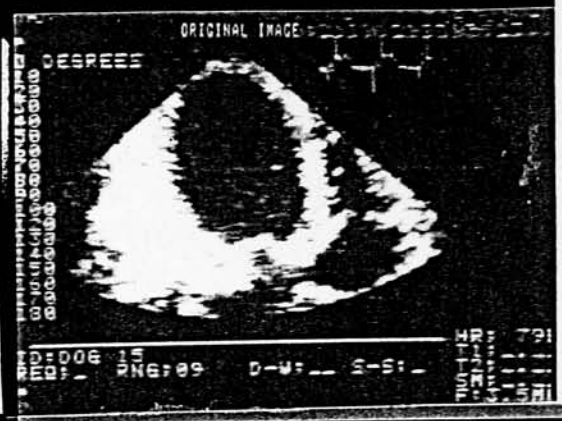
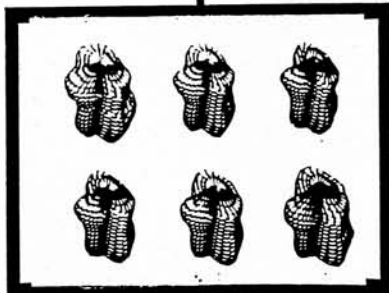
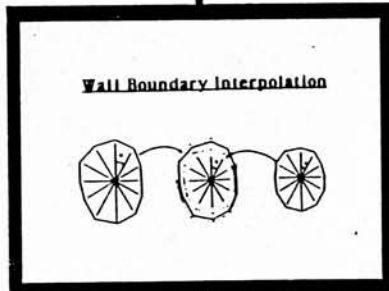


Figure 3 - Four-dimensional reconstruction steps.

Although it is entirely possible to produce the reconstructions by hand tracing the left ventricle boundaries, to achieve any level of practicality and accuracy requires the automation of the boundary tracing process. Aside from the obvious fact that operator interaction is tedious and time consuming, the accuracy and reproducibility of the manual traces also carry variabilities due to operator subjectivity. This effort addresses the need for an automated process of boundary detection. The conversion from training a human operator to trace the images to programming a computer to accomplish the same task is not a trivial one. The image processing algorithms presented by Schott, 1987 proved to be adequate when a human observer performed the tracings. To train a computer to do the same task, however, requires a larger and entirely different combination of image processing algorithms. The human visual system, unlike computers, can globally process images and trivially detect borders while ignoring certain image information (e.g. speckle noise in echo images). In fact, it is relatively easy for a trained observer to interpolate missing image information (e.g. dropouts in echo images due to degradation of the ultrasound beam) because of the high-level image processing coupled with an even higher-level knowledge base. Computer processing, on the other hand, can process images only on a pixel-by-pixel basis with some limited processing on groups of pixels (kernels). Because of this limited scope, computer processing is very sensitive to noise in the image. This noise, if severe enough, can easily be misclassified as a boundary feature. A summary of the algorithms addressing the issues stated will be described in the next section with some statement and treatment of the conditions and criteria that guided the choice of algorithms and the design of their development.

1.3 Computer Software/Hardware Overview:

The objective of this project is directed at the selection of a combination of algorithms and data structures which exploits the special hardware capabilities of the Gould/Deanza IP8500 Image Processing System to the solution of the boundary detection problem. At the same time, considerations for software flexibility have been incorporated in the program code to allow portability to other machines. The optimum configuration, however, resides with the exotic hardware of the IP8500 system. Because of its highly specific processing power, a section is devoted to describing the details of the IP8500 and its components. This will clarify many of the rationales behind the algorithm logic and data structure design. A brief overview of the algorithm flow is discussed which leads to individual algorithm subsections. The algorithm subsections will discuss in detail the image preprocessing, threshold selection, quad-tree image data structure variation, contour following, and an implementation and application of binary morphological operations. A section discussing the integration and performance results of the algorithms is also presented along with recommendations for improvements and further areas of study.

A note about the selection of algorithms. Because a preference has been made for a specific hardware system, certain classes of image processing algorithms become less suited to the overall boundary tracing solution because of their computational complexity and non-linear nature. An example of such an algorithm is the standard deviation kernel. The standard deviation kernel is a commonly used kernel operation for determining areas having large grey-level variations. This process involves calculation of the grey-level standard deviation of the image area encompassed by the kernel as it is moved over the entire image. This creates a standard deviation map giving a measure of overall image brightness variability. With this process it is possible to use this kernel to determine the location of boundary dropouts caused by a weak signal as well as detect edges representing points of transition between tissue and chamber. The drawback of this algorithm, however, lies in the non-linear nature of the pixel-by-pixel calculations disallowing parallel image computations to be implemented. This in turn magnifies the computational complexity making these types of algorithms even less favorable to implement as a tool for the solution. The parallel image operation concept is a recurring theme throughout this treatise. Although it was not always possible, efforts were made to adhere to this paradigm in implementing the algorithms.

2. Image Processing Hardware:

The IP8500 is a very powerful image processing system which has been one of the workhorses of the image processing community for almost a decade. In its fullest configuration, it can contain a myriad of component combinations consisting of several 1024 x 1024 x 8-bit memory planes, video digitizers, interactive devices, and specialized image manipulation hardware. The modularity and programmability of its design allows it to accomplish an even greater combination of image manipulating operations which is limited by the user's intimate knowledge of the hardware and the registers that control them. As with many systems which attempt to combine general modularity with specialized hardware to meet user specific needs, the result is a system which is notoriously difficult to program. The IP8500 is no exception.

The specific IP8500 system on which the algorithms were developed and implemented is housed and maintained at the Digital Imaging and Remote Sensing Laboratory. The overall image processing system has been partitioned into three workstations consisting of a computer terminal, an interactive joystick peripheral, a high resolution color monitor, and hardware resources from the IP8500 as illustrated in **Figure 4**.

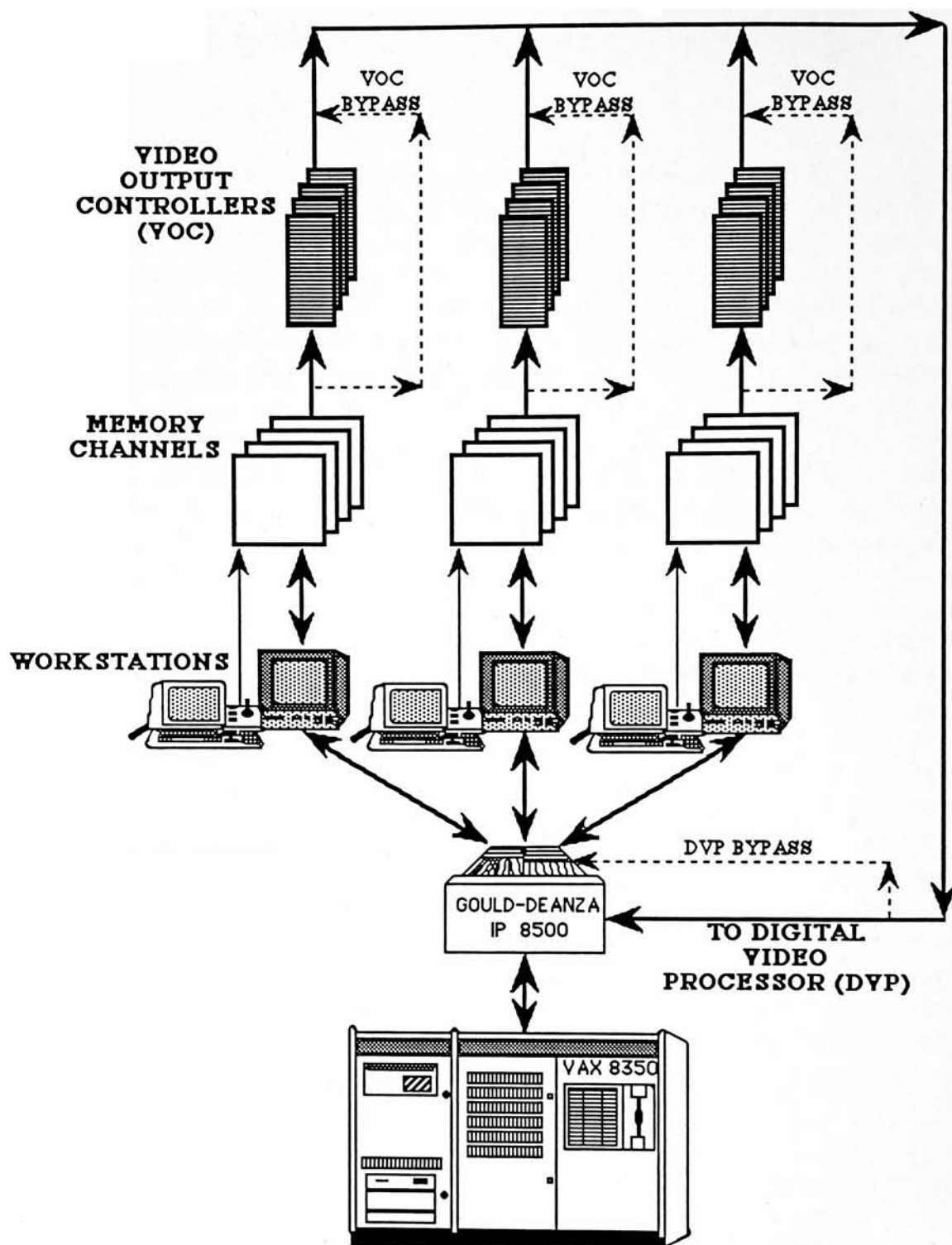


Figure 4 - Overall image processing system.

The IP8500 resources allocated for each workstation unit, consisted of a bank of image memory channels and a Video Output Controller (VOC) as illustrated **Figure 5**. The image memory channels are physically arranged into four 1024 pixels by 1024 pixels by 8-bit pixel image planes which are normally configured to operate as either four separate 8-bit monochrome images, two 16-bit monochrome images or one 24-bit color (8-bit red, 8-bit green, 8-bit blue) image with 8-bit graphics overlay. For the purposes of this project, the former two memory arrangements were employed by the algorithms. In addition, each of the channels can be scrolled and zoomed in relation to a given origin with the aid of special zoom/scroll registers. This hardware scroll feature is an essential manipulation that gives the system an extremely fast capability to spatially shift the image channels relative to each other -- a primitive operation in spatial convolution and morphological operations of images. Also included in the image memory resource are grey-level manipulating sections called Intensity Transformation Tables (ITT). These allow all the image pixels of a given grey-level to be mapped and changed to another grey-level in a look-up table (LUT) fashion. These hardware ITT's render grey-level transformations such as grey-level thresholding and histogram manipulations a trivial process. The other major IP8500 resource in a workstation is the video output controller (VOC) which controls how the image data is to be displayed on the video monitor including image channel selection, color representation/manipulation, and overlay graphics control of the graphics channel, programmable cursors, and alphanumeric generator. The workstation system configuration described thus far is basically limited to image display with some limited spatial and grey-level manipulation. The power of the IP8500 system comes in the form of a shared resource called the Digital Video Processor (DVP).

The DVP is conceptually the workstations ALU in the sense of image data sets, i.e., the DVP operates on the images in the memory channels in much the same way an ALU for a serial central processing unit operates on single registers. The DVP can perform arithmetic operations on images such as add, subtract, multiply, divide, and compare. In addition, it can perform Boolean operations on images such as NOT, AND, OR, XOR and bit-wise operations such as logical/arithmetic shifts and rotates. The specific DVP in this system is limited to processing images having pixel dimension of 512 x 512 pixels. Nominal execution time of arithmetic binary operations on 512 x 512 x 8-bit operand images is 0.033 seconds or 30 operations cycles/second which in terms of video rates (30 Hz) is for all practical purposes a real-time process. The DVP operates on the memory channels in a pipe-line fashion sending the pixel data rapidly through two levels of multiple ALU banks and a data comparator. The path of the data can be configured to achieve either 8 or 16 bit operations with more complex operations requiring several passes through the DVP. In addition to the DVP, there is a special histogramming board which can be used in conjunction with the DVP for high speed calculation of a 256 grey-level histogram. The latter hardware component will be essential to the fast execution of an algorithm to be discussed later.

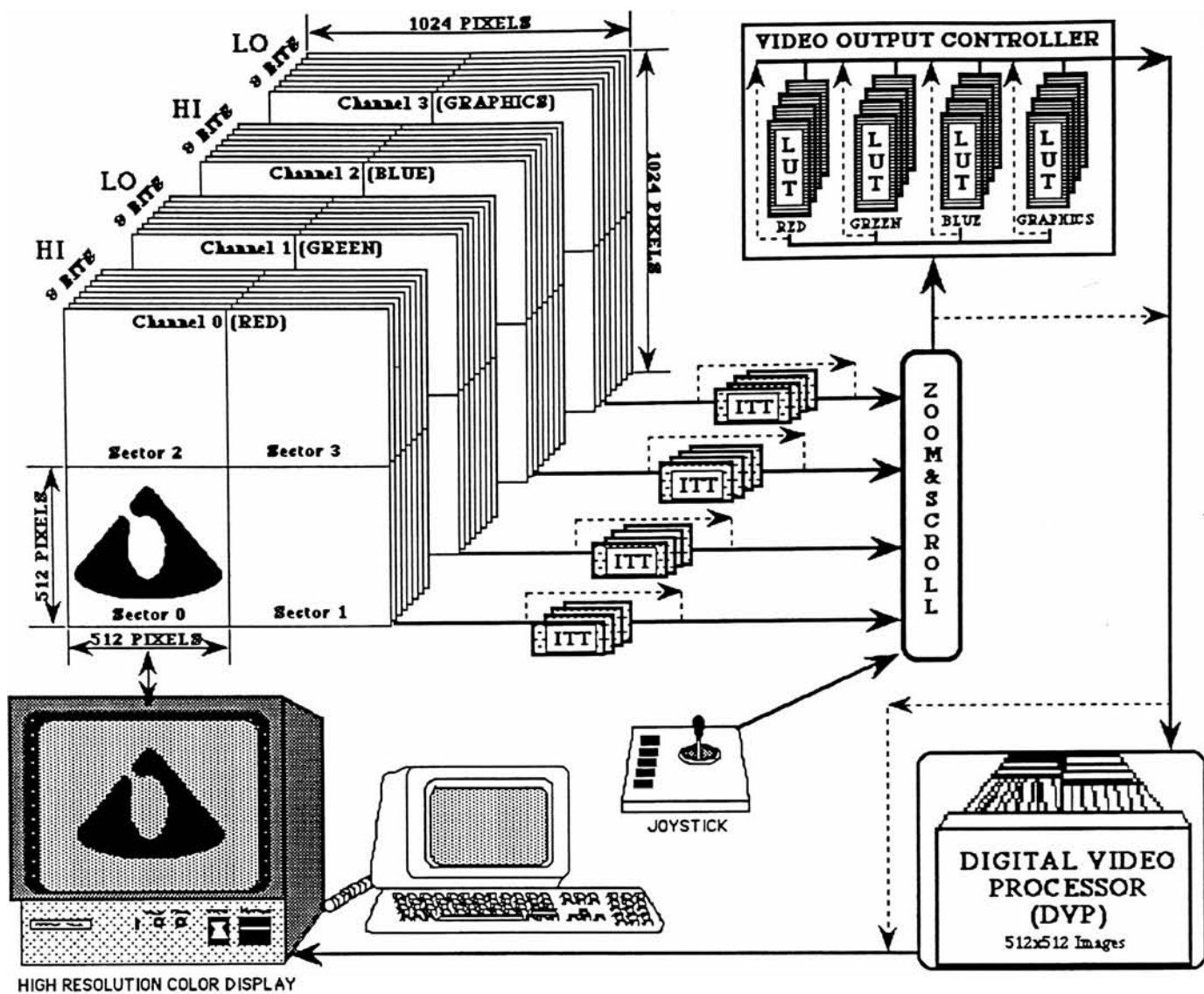


Figure 5 - Workstation hardware configuration.

3. Image Processing Software:

There are several ways to access control of the IP8500. At the most primitive level, the user may control the image processing system via bit manipulation of special hardware registers to configure data paths and control execution. This level of programming interface is referred to as LEVEL 0 which provides the greatest flexibility and control but at the expense of programming ease. It requires the user to be intimately familiar with the image processing hardware and all the registers that control all the individual components in much the same way a user is required to be very knowledgeable of the architecture of a micro-coded processor. A higher level of interfacing with the system involves an abstraction of the software into the major components of the system in roughly the same manner in which it was described in the previous section. This LEVEL 1 interfacing groups the software calls at a more intuitive level to allow the user to focus on programming the system based on the major system components and the corresponding operation each one accomplishes. Housekeeping responsibilities are also shifted away from the user which, at the register level, requires user programming to maintain. The interfacing software was supplied by DEANZA at both programming levels with FORTRAN as the binding language. The actual interfacing software used, however, was a customized package (referred to as **IPI** for **I**mage **P**rocessing **I**nterface) developed at DIRS/CIS by Steven L. Schultz. This package, having a LEVEL 1 programming functionality, was used in place of the OEM software because it has been tailored to meet specific needs of the lab as well as implementing several optimizations of LEVEL 0 type calls.

The final algorithm software was developed using both FORTRAN and C programming languages. FORTRAN routines involving the IPI routines were developed to provide another layer of abstraction to the user giving the image processing calls a degree of device independence at the higher programming level. Another motivation for using FORTRAN comes from the ease of linking to other useful routines previously developed in FORTRAN. The C programming language was used at the algorithmic level to allow for more elaborate data structuring and recursion. A link was then made between the two languages to create the executable code. The following sections will now describe these algorithms.

4. Algorithms:

The set of image processing algorithms used to solve the boundary detection problem can be summarized by the following block diagram Figure 6.

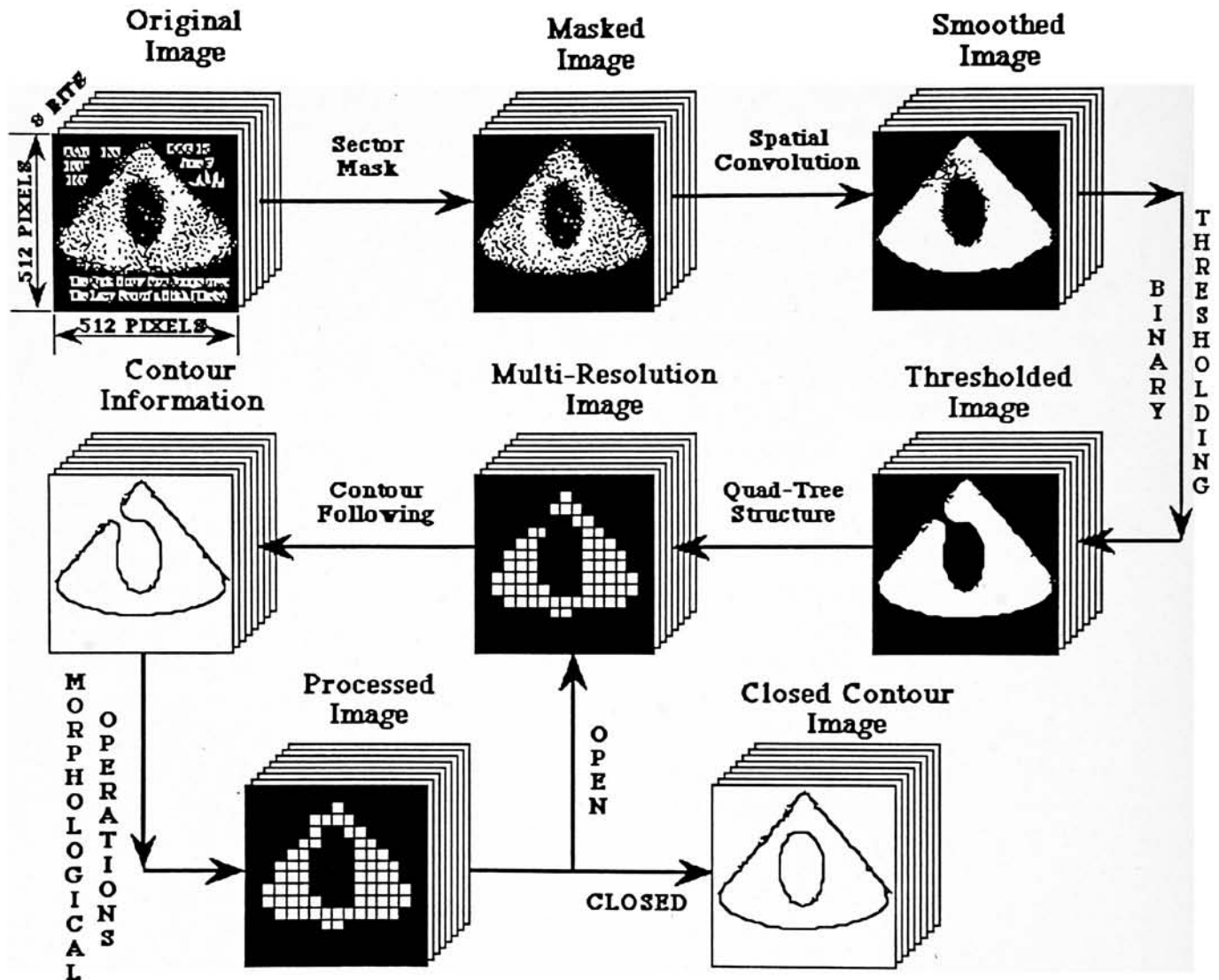


Figure 6 - Sequence and flow of algorithms.

The first two processes, sector region of interest and spatial convolution, are preprocessing steps which are necessary operations to limit the 512 x 512 pixel image area to a smaller region of interest and to reduce the amount of noise in the image. The thresholding step utilizes a special algorithm to generate a simple statistic for selection of a grey-level thresholding without recourse to a histogram. After thresholding, the image is transformed into a variant of a quad-tree data structure. This data structure was designed to emulate a quad-tree multiresolution structure of the image without the search constraints that come with traditional quad-tree structures implemented using linked lists. The resulting multiresolution representation is then subjected to a contour detection algorithm which attempts to determine the contour of the binary image at select resolutions. This step will detect any openings caused by loss of resolution leading to the incomplete delineation of the endocardial boundary. This is due to the parallel (rather than a perpendicular) interrogation of the left ventricular endocardium by the ultrasonic signals. This is a critical step because it determines whether or not an opening has occurred and at what resolution level it became apparent to the algorithm. This information gives some mensuration of the greatest dimension of the structuring element needed to morphologically close the boundary opening. The success of the closing operation can then be monitored by the contour detection algorithm again and then subjected to another closing operation if necessary. This feed-back loop allows the algorithm to cycle through the process until a successful closing is achieved. The succeeding sections will elaborate on the algorithms, design motivations, and special features.

4.1 Sector Region of Interest:

Sector region of interest defines the boundaries in which the images of the left ventricle are expected to coincide. The shape of this boundary represents the field-of-view sweep of the sonogram sector scan. Outside this boundary is sonogram generated annotation data used for displaying and archiving patient/equipment setting information. Masking the original image with the hand drawn sector mask removes all data outside of this region and limits the search space to relevant echo image data. This masking operation is quickly accomplished using the bitwise AND operation native to the IP8500's DVP instruction set and is diagrammed in **Figure 7**.

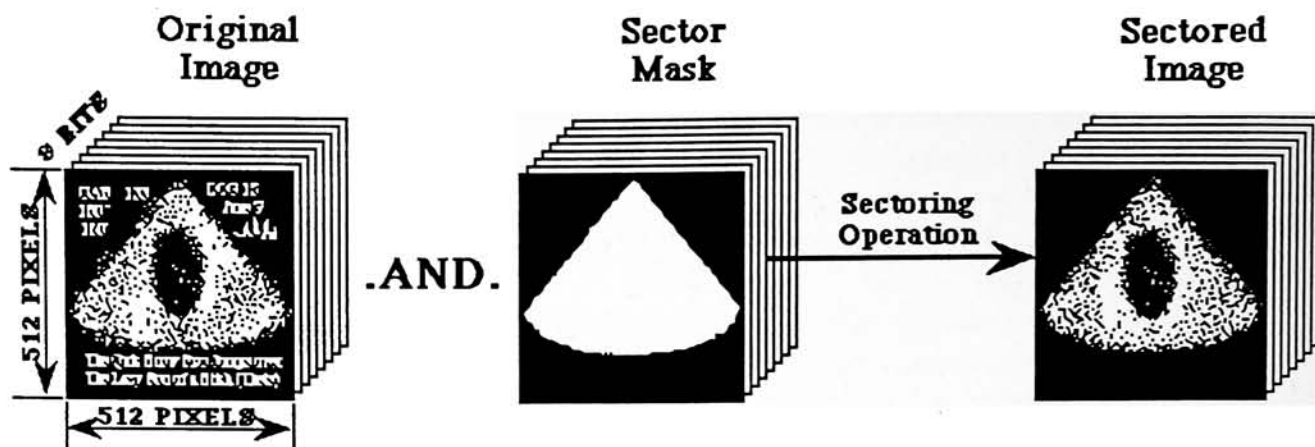


Figure 7 - Sector mask operation.

4.2 Spatial Convolution:

The next preprocessing step is a smoothing spatial convolution which limits the speckle noise present in an image. Figure 8 diagrams this process.

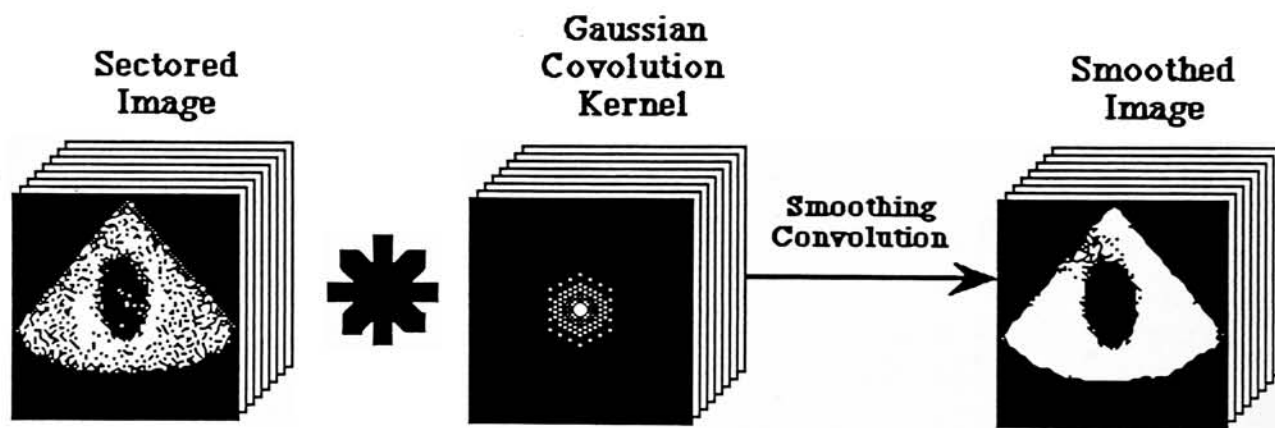


Figure 8 - Smoothing Convolution.

To understand this process, a reference needs to be made to linear systems theory and the concept of a Fourier Transform of an image. Simply stated, the Fourier transform of an image is the result of a mathematical transformation that describes the image in terms of component harmonic periodic sinusoidal brightness functions of varying amplitudes, i.e., a spectrum of spatial frequencies. The following illustration (Figure 9) consisting of images of vertical alternating bands of low and high brightness levels conveys the concept of varying spatial frequencies.

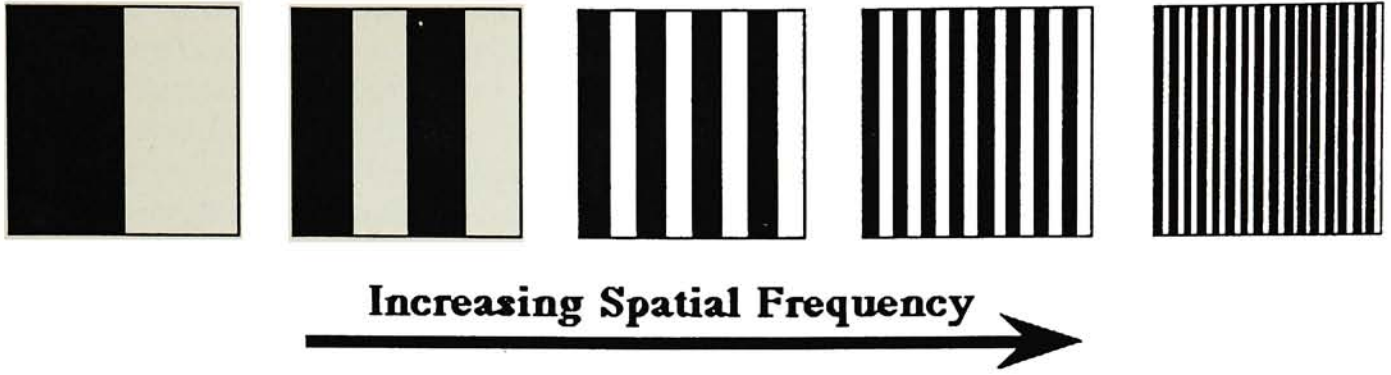


Figure 9 - Varying spatial frequency in the horizontal direction.

The spatial frequency representation of $\text{image}(x,y)$ is given by the following complex valued transform equation.

$$\text{IMAGE}(\xi, \eta) = \iint_{-\infty}^{\infty} \text{image}(x, y) e^{-i2\pi(x\xi + y\eta)} dx dy$$

$\text{IMAGE}(\xi, \eta)$ is the Fourier transform of the image. The contribution of the different spatial frequencies dictate the overall characteristics of the image. A one-dimensional analog of this concept can be observed in acoustics in which sound signals can be decomposed into component harmonic frequencies. A sound signal which has a large low-frequency component is characterized by a signal that changes slowly as a function of time. As an example, certain speech pronunciations such as "o" in "low" and "u" in "put" have a significant low frequency sound component whereas fricatives such as "s" in "hiss" tend to have high-frequency contributions. In the case of two-dimensional images, low spatial frequency sinusoidal brightness components carry the information determining overall shape and global brightness levels resulting in an image characterized by slowly varying brightness levels as a function of space. High spatial frequency information, on the other hand, contains the fine image detail such as edges and local brightness variations perceived as texture. The speckle noise that is characteristic of echo images falls under the category of high frequency information and can be thought of as the two-dimensional analog of "static noise" that is commonly found in acoustics and communications.

Removal of this type of high-frequency noise is usually accomplished through a process known as low-pass filtering in which all signal information below a given cutoff frequency is maintained (passed through) and all frequencies above the cutoff is attenuated or removed completely. Low-pass filtering can be achieved in two ways - through frequency domain filtering or spatial domain convolution. Frequency domain low-pass filtering involves multiplying the spectrum of the image by a filter function which attenuates the power of the

spectrum as the frequency increases. This frequency filtering concept is analogous to modifying a high fidelity stereo signal through a graphic equalizer by the setting of slider bars to attenuate the selected frequency bandpasses. Spatial domain filtering, on the other hand, uses a weighting function or, more accurately, a point spread function (psf) to respecify the neighboring image area contributions to the brightness of each image point. This process (by which the optical behavior of lenses are also described) can be described mathematically by a convolution given by the following equation.

$$\text{filtered_image}(x,y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \text{image}(x,y) \text{psf}(x - \alpha, y - \beta) d\alpha d\beta$$

Convolution in terms of digital images follows the same concept of assigning to each image point a weighted contribution of neighboring image points based on the mirror image transformation of a given kernel. If a kernel is used without the mirror transformation, then the process is called a correlation. If the kernels are radially symmetrical, then the convolution and correlation process will yield the same results. The operations, in any case, is a discrete process. The nomenclature describing convolution also changes to convey the discontinuous property of the image and psf. Image points are quantized into pixels and the point spread function is represented by a discrete approximation known as a kernel. The convolution equation in its discrete form is given by the following equation.

$$\text{filtered_image}(x,y) = \sum_{i=-\frac{M}{2}}^{\frac{M}{2}} \sum_{j=-\frac{N}{2}}^{\frac{N}{2}} \text{image}(x + i, y + j) \text{kernel}(i,j)$$

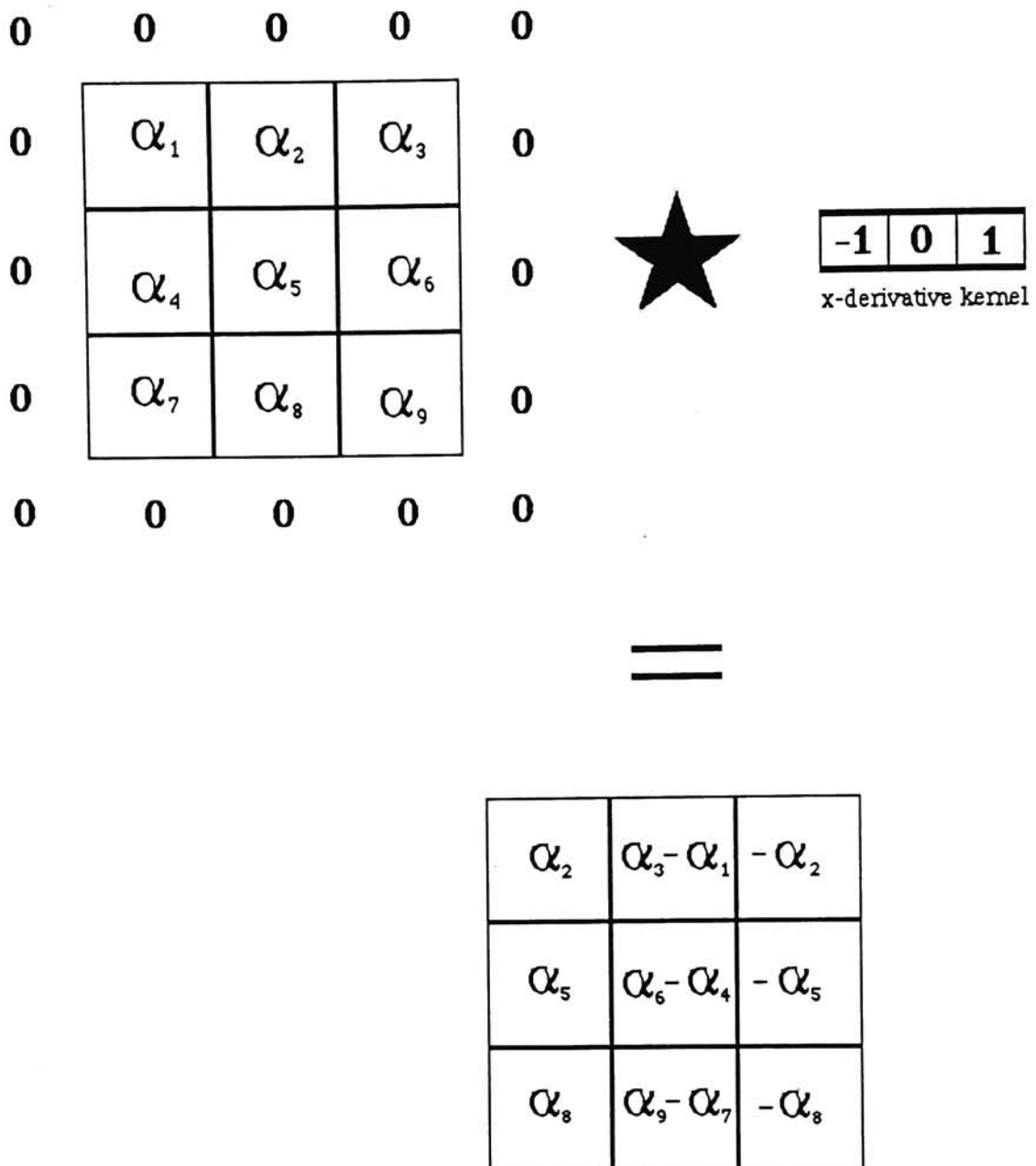
Where $M \times N$ represent the kernel dimensions in pixels, x and y are the image pixel locations with respect to the image coordinates, and i and j are the location of the kernel pixels with respect to the kernel coordinates.

According to linear systems theory, there exists a duality between the two processes and that each has a corresponding representation in the other domain, i.e., a convolution in the spatial domain can be represented as a multiplication in the frequency domain and a multiplication in the spatial domain can be represented as a convolution in the frequency domain. Therefore, a convolution of an image using a specified kernel generates the same result as Fourier transforming both the image and kernel into frequency space, multiplying the two transforms, and back transforming the product to the spatial domain. This relationship is expressed concisely in the following relationship.

$$f(x, y) * * g(x, y) \Leftrightarrow F(\xi, \eta)G(\xi, \eta)$$

The above expression states that there exists two fundamental ways in which the desired filtering can be accomplished. The computational economy of the two methods depends on the type of hardware available for the calculations. In the case of serial computations made on general purpose computers, the frequency domain filtering algorithms have been favored over spatial convolutions for images of any significant size in terms of spatial and brightness resolution. This is in light of the development of the Fast Fourier Transform (FFT) algorithm (Cooley 1965) resulting in a numerical method optimization of the Discrete Fourier Transform (DFT). The FFT was deemed the more computationally economical algorithm in comparison to spatial convolution for serial computations.

In cases where special hardware such as the IP8500 is available, spatial convolution becomes a fast parallel operation rivaling the FFT algorithm. This parallelism is possible because of the shift invariant properties of the kernel operation, i.e., all the calculations involving the convolution kernel as a weighting factor is applied equally to each and every pixel in an image. The ability of the IP8500 to process the images in parallel reduces convolution as well correlation operations to a series of image scrolls, image multiplications, and image additions. The number of operations is dependent only on the size of the kernel and not the size of the image (up to 512x512 pixel dimensions). To appreciate the enhancement that the hardware offers in a correlation or convolution calculation, consider a simple case of a 3x3 image correlated with a 1x3 kernel illustrated **Figure 10** (*N.B.* a convolution would flip the kernel from **-1 0 1** to **1 0 -1** before applying the kernel whereas a correlation would use the kernel form **-1 0 1**).



**Figure 10 - Serial computation example of correlation
(or convolution).**

Note that undefined pixels outside of the image domain are assumed to have a zero pixel value. From a serial computation paradigm, the convolution of the kernel with the image involves localizing the calculation to the pixels of interest superimposed by the kernel as it is translated pixel-by-pixel over the entire image. For each pixel coincident with the center of the kernel, the calculated value is deposited into the corresponding center pixel in a result image. The final

result image after the kernel has been applied to all pixels of the image is shown above. One only has to increase the image size to appreciate the computational demand for this process. The alternative to the conventional serial calculation involves the use of parallel hardware. The parallel computation method made possible by the IP8500 follows a different approach. Since the DVP applies the calculations at the image level (i.e., all calculations are made to all the pixels in the image eliminating the need to address each pixel individually as in the previous approach), it is possible to represent the calculation as a sum of a series of translated and kernel weighted images. **Figure 11** diagrams this process for the same 3x3 image and 1x3 kernel and verifies the equality of the results with the pixel-by-pixel computation method.

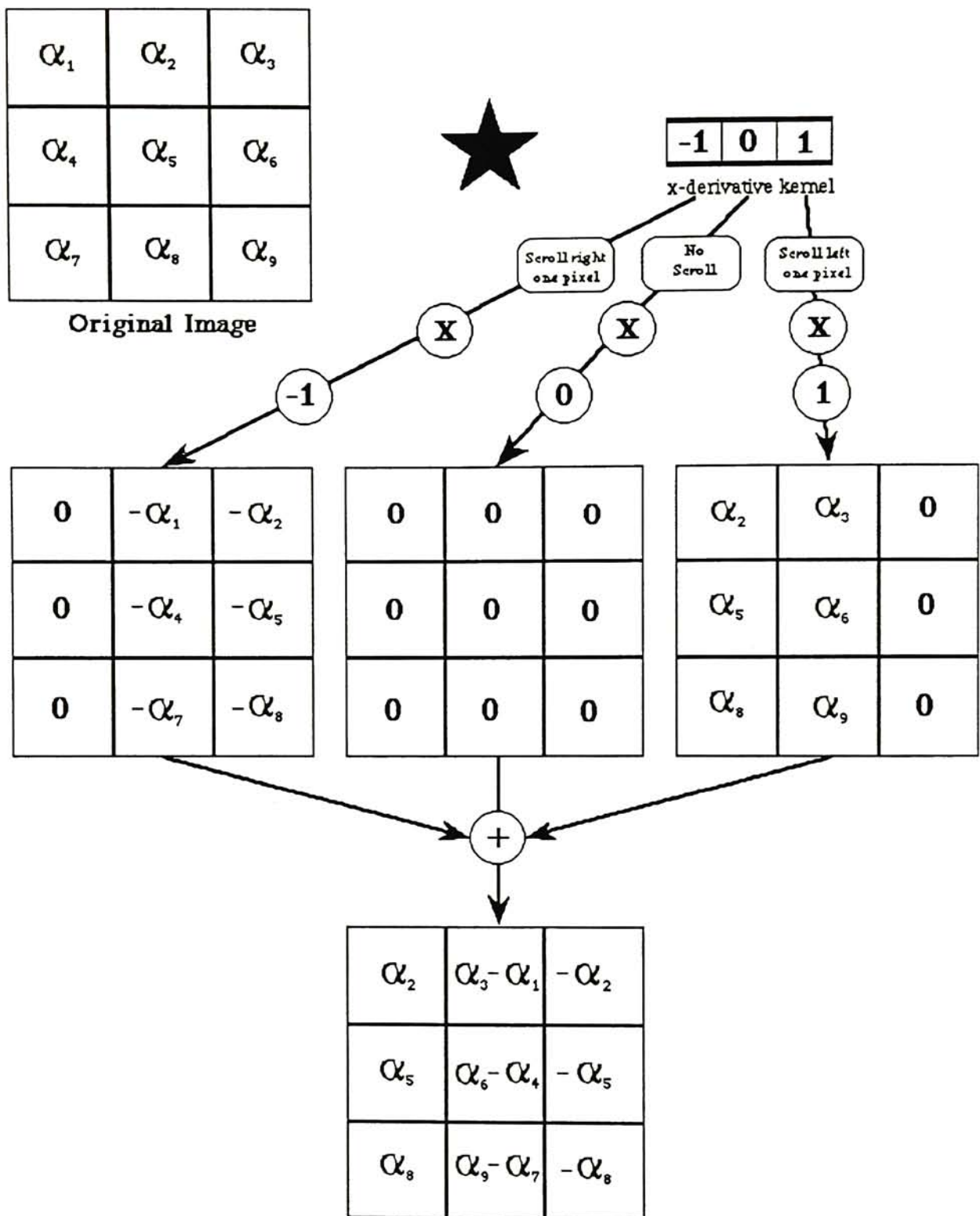


Figure 11 - Parallel computation example of correlation (or convolution).

For this specific kernel, the IP8500 process requires 3 image scrolls, 3 image by constant multiplications, and 3 image additions. This can be further optimized to 2 image scrolls, 1 negation, and 1 addition. In general, a kernel of $N \times N$ pixel size will require at most $N^2 - 1$ image scrolls, N^2 multiplications by a constant, and N^2 additions. An additional scaling step is also added to limit the grey-level range to the range of the memory channels.

The specific kernel chosen to low-pass filter the image was a 15×15 Gaussian convolution kernel shown below in **Figure 12**

2	2	3	4	5	5	6	6	6	5	5	4	3	2	2
2	3	4	5	7	7	8	8	8	7	7	5	4	3	2
3	4	6	7	9	10	10	11	10	10	9	7	6	4	3
4	5	7	9	10	12	13	13	13	12	10	9	7	5	4
5	7	9	11	13	14	15	16	15	14	13	11	9	7	5
5	7	10	12	14	16	17	18	17	16	14	12	10	7	5
6	8	10	13	15	17	19	19	19	17	15	13	10	8	6
6	8	11	13	16	18	19	20	19	18	16	13	11	8	6
6	8	10	13	15	17	19	19	19	17	15	13	10	8	6
5	7	10	12	14	16	17	18	17	16	14	12	10	7	5
5	7	9	11	13	14	15	16	15	14	13	11	9	7	5
4	5	7	9	10	12	13	13	13	12	10	9	7	5	4
3	4	6	7	9	10	10	11	10	10	9	7	6	4	3
2	3	4	5	7	7	8	8	8	7	7	5	4	3	2
2	2	3	4	5	5	6	6	6	5	5	4	3	2	2

Figure 12 - 15×15 Gaussian convolution kernel

This particular kernel was selected because it is a radially monotonically decreasing function which translates into a monotonically decreasing weighting of pixels. The Gaussian kernel also has the unique property of having another Gaussian function as its Fourier transform pair. This means that in the frequency space, the Gaussian low-pass filter has a monotonically increasing attenuation effect as the spatial frequency increases. This monotonically increasing attenuation property eliminates the phenomenon called "leakage" or "ringing" which introduces spurious high frequency artifacts. Overall, the Gaussian function provides good blurring characteristics to remove noise, average brightness areas, and limit image detail. The resulting image from this process is a very smoothed image which can now be thresholded. As an aside, it should be noted that although the information in the form of speckle noise is not usable in this context, it does hold information related to tissue characteristics (Zhu 1990) based on scattering properties. Future refinements of extracting information from the speckle information may be of significant use to improve the performance of the existing algorithms.

4.3 Robust Automatic Threshold Selection:

The next algorithm needed to segment out the boundaries is an automatic thresholding routine. Thresholding or grey-level binarization is a simple operation in which grey-levels above a selected threshold are set to the brightest digital count (255) and those grey-levels below set to the darkest digital count (0) (Gonzalez, 1977). This reduces the grey-level dimensionality to a binary image and segments the image by maximizing the contrast between the tissues and the chamber. This creates a simpler scenario for a contour following algorithm to determine the location of the border. Thresholding is a trivial algorithm to implement, but the automatic selection of the proper threshold is not. The common approach to automatic threshold selection involves the computation of the grey-level histogram of an image and the selection of an intermodal grey-level of a bimodal histogram distribution. The threshold value of choice is often the grey-level midway between the peaks of the two maximum points representing the image and background pixels in the histogram as shown in Figure 13.

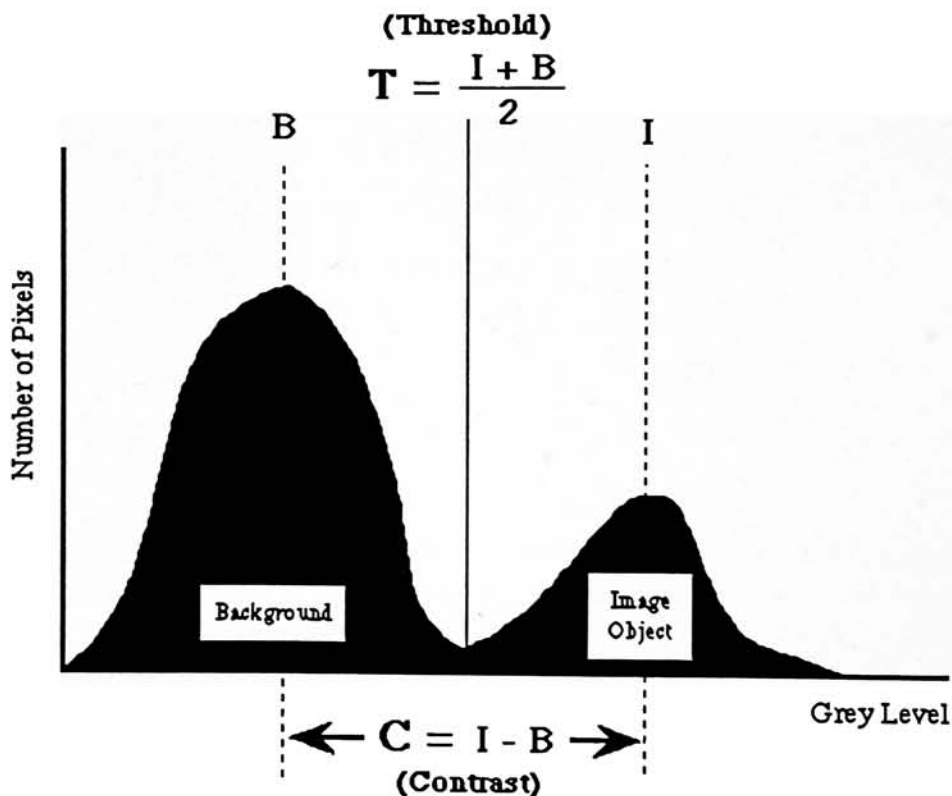


Figure 13 - Histogram based grey level threshold selection.

Bimodal distributions, however, do not always occur for high contrast object/background images. Images many times generate unimodal or even multimodal distributions due to brightness gradients and noise. In these situations, histogram statistics become an inadequate

criteria because of the difficulty in resolving the proper threshold value. Histogram statistics also have the disadvantage of being sensitive to the sampling size and the ratio of image pixels to background pixels. This results in statistical fluctuations making automatic threshold selection using histogram statistics inaccurate.

A threshold selection method developed by Kittler *et al.* 1985 called the **Robust Automatic Threshold Selector (RATS)**, bases its threshold selection on the ratio of the **sum of grey-grad** values to the **sum of grad** values. The **grad** value is defined as the resulting maximum value between the x-derivatives and y-derivatives for each pixel in an image. The **grey-grad** values are simply the original grey value of the image multiplied by the grad values. The insensitivity of this statistic to image size eliminates the inadequacies inherent with histogram techniques operating on non-bimodal histograms. The RATS algorithm has been tested successfully by its developers in assembly line scenarios involving high contrast images of machine components. The high contrast nature of the echo images prompted the selection of this algorithm to threshold the images in this situation. This particular method was also chosen because the intermediate operations to determine the appropriate threshold lends itself well to the architecture and capabilities of the IP8500 system.

Theory for this technique stemmed from previous work by Kittler, 1983 involving absorption edge detectors. These edge detectors have a special property of generating edge values proportional to the image/background contrast invariant of edge position and rotation. These statistics were based on the observation that the sum of edge magnitude value of an edge operator (kernel) in the vicinity of a scan line intersecting a vertical edge is a constant. The following is a detailed treatment of the theory behind the this concept. Consider a simple scene with a bright image having a mean grey-level value **I** against a dark background having a grey-level value **B**. The contrast between the image and the background can be expressed as

$$C = I - B$$

Now consider a horizontal scan line at the vicinity of an edge and the grey-levels associated with the scan line can be represented by **Figure 14**.

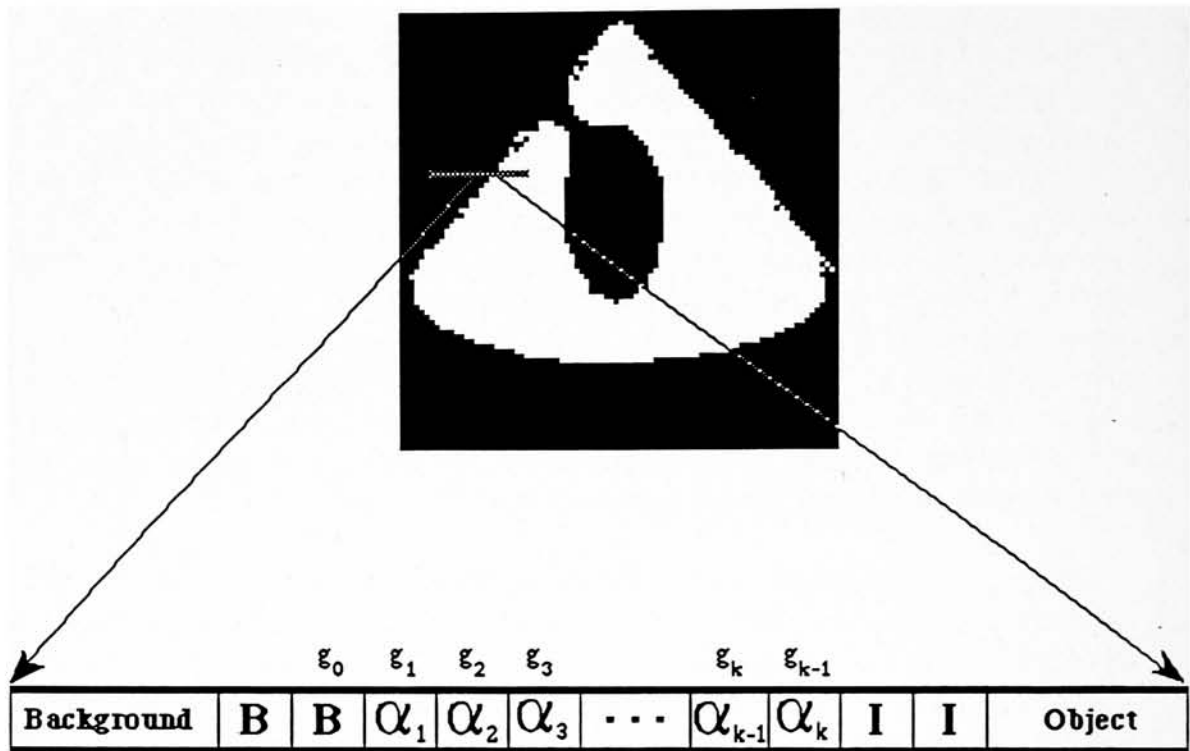


Figure 14 - Edge scan line (grey).

By applying a 1x3 x-derivative kernel along the segment of the scan line as shown in Figure 15,

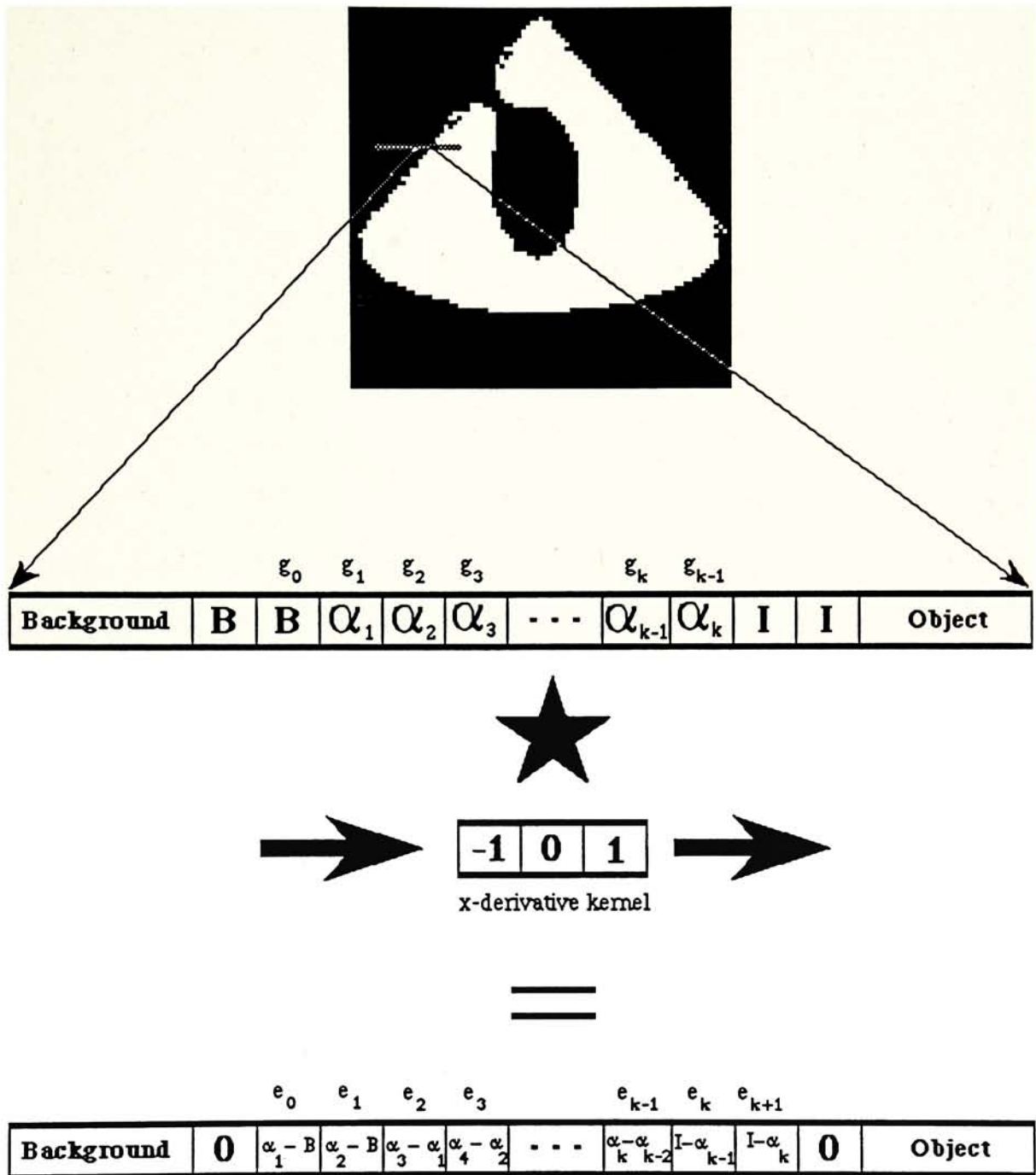


Figure 15 - Edge scan line derivative (grad).

the sum of the resulting derivative terms can be written as follows.

$$\sum_{j=0}^{k+1} e_j = \begin{pmatrix} (\alpha_1 - B) + \\ (\alpha_2 - B) + \\ (\alpha_3 - \alpha_1) + \\ (\alpha_4 - \alpha_2) + \\ \vdots \\ (\alpha_k - \alpha_{k-2}) + \\ (I - \alpha_{k-1}) + \\ (I - \alpha_k) \end{pmatrix} = 2(I - B)$$

The term e_j represents the grad value generated by the derivative kernel. By substituting the equation $C = I - B$, the result can be expressed below.

$$\sum_{j=0}^{k+1} e_j = 2C$$

The same concept holds true for an image to background transition giving a $-2C$ value instead of a $2C$ which merely indicates a contrast reversal, i.e., a transition from the bright object pixels to the dark background pixels. By taking the absolute value of the edge operations e_j , the direction dependence of the derivative operation is removed.

As a result, the sum of the derivative values along a scan line crossing a single edge can be generally expressed by the following.

$$\sum_{j=1}^N |e_j| = 2C$$

The parameter N is the number of pixels processed by the x-derivative kernel. Each scan line, however, can cross several edges which expands the equation.

$$\sum_{j=1}^N |e_j| = 2Cn$$

The term n is the number of edges crossed by the scan line. This expression shows that the sum of the vertical edge magnitude is proportional to the contrast by the number of edges crossed by the scan line.

Assuming the image has a dimension of $N \times N$ pixels yielding N scan lines N pixels long, the above expression can be augmented to relate the grad value sum relation to contrast for a two-dimensional image.

$$\sum_{i=1}^N \sum_{j=1}^N |e_{ij}| = \sum_{i=1}^N 2Cn_i = 2C \sum_{i=1}^N n_i = 2Cm$$

In the above equation, e_{ij} represents the derivative magnitude of the j th pixel of the i th scan line, n_i is the number of times the i th scan line crosses an edge, and m is the total number of edge pixels in the image. As with the individual scan lines, the sum of the edge magnitude for the whole image is also proportional to the contrast by the number of edges crossed by all the scan lines in the image.

A similar treatment can be made for the horizontal edge case using a 3×1 kernel to scan the image vertically to detect horizontal edges. The results are analogous to the vertical edge case, i.e., the sum of the horizontal edge magnitudes is proportional to the contrast by the number of horizontal edges crossed by the vertical scan lines. The separate vertical and horizontal cases can be combined by taking the maximum edge magnitude for each pixel, i.e., for each pixel in an image, the horizontal and vertical edge magnitude are compared to determine what type of edge (orientation) the pixel represents. By taking the larger of the two magnitudes, a combination of the horizontal and vertical edge values can be merged to create a representative derivative map from which the grad statistic is formulated. The working formula for calculating the sum of the grad value is given by the following equation.

$$\sum_{i=1}^N \sum_{j=1}^N |e_{ij}| = \sum_{i=1}^N \sum_{j=1}^N \max\{|e_{ij}|_x, |e_{ij}|_y\} = 2Cm$$

Where $|e_{ij}|_x$ and $|e_{ij}|_y$ represent the edge magnitude generated by the x-derivative and y-derivative kernels, respectively. This represents the sum of the grad value of the image as described by Kittler. An improvement implemented to this algorithm is a simple inclusion of possible contributions by edges detected by diagonal kernels in addition to the previous two kernels discussed. The resulting gamut of derivative kernels used in the grad value calculation is illustrated below in **Figure 16**.

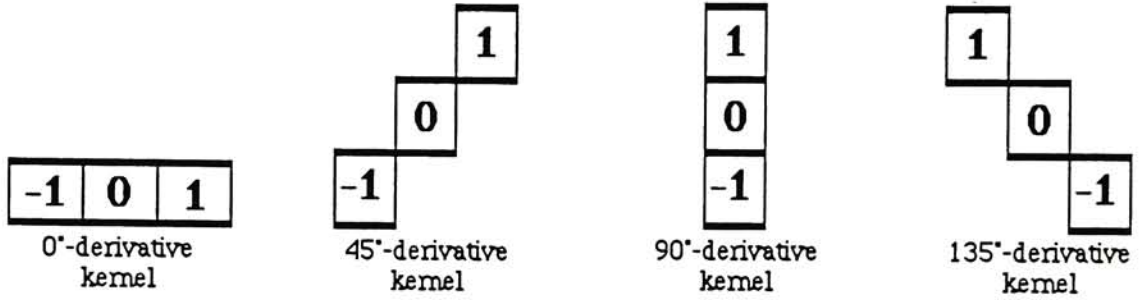


Figure 16 - Derivative kernels.

For each pixel location, the sum of grad values now consists of the maximum edge values selected from each possible orientation (as allowed by the pixel tessellation). By including the other possible edge contributions, a more accurate edge map can be derived resulting in a more representative sum of grad statistic. This is especially significant since the images of interest have edge contributions for all possible orientations because of the elliptical arrangement of the boundaries being examined. The final working equation for the sum of grad statistic is given by the following.

$$\sum_{i=1}^N \sum_{j=1}^N |e_{ij}| = \sum_{i=1}^N \sum_{j=1}^N \max\{|e_{ij}|_{0^\circ}, |e_{ij}|_{45^\circ}, |e_{ij}|_{90^\circ}, |e_{ij}|_{135^\circ}\} = 2Cm$$

The terms $|e_{ij}|_{0^\circ}$, $|e_{ij}|_{45^\circ}$, $|e_{ij}|_{90^\circ}$, and $|e_{ij}|_{135^\circ}$ represent the edge magnitudes generated by the kernels illustrated in **Figure 16**.

The next statistic to be derived is the grey-grad value. This statistic is derived from the sum of the product of the grey-level values of the original image and the derivative map described above. By reverting back again to the single scan line case, this relationship can be illustrated by the same scan line method presented for the grad statistic case as shown by the **Figure 17**.

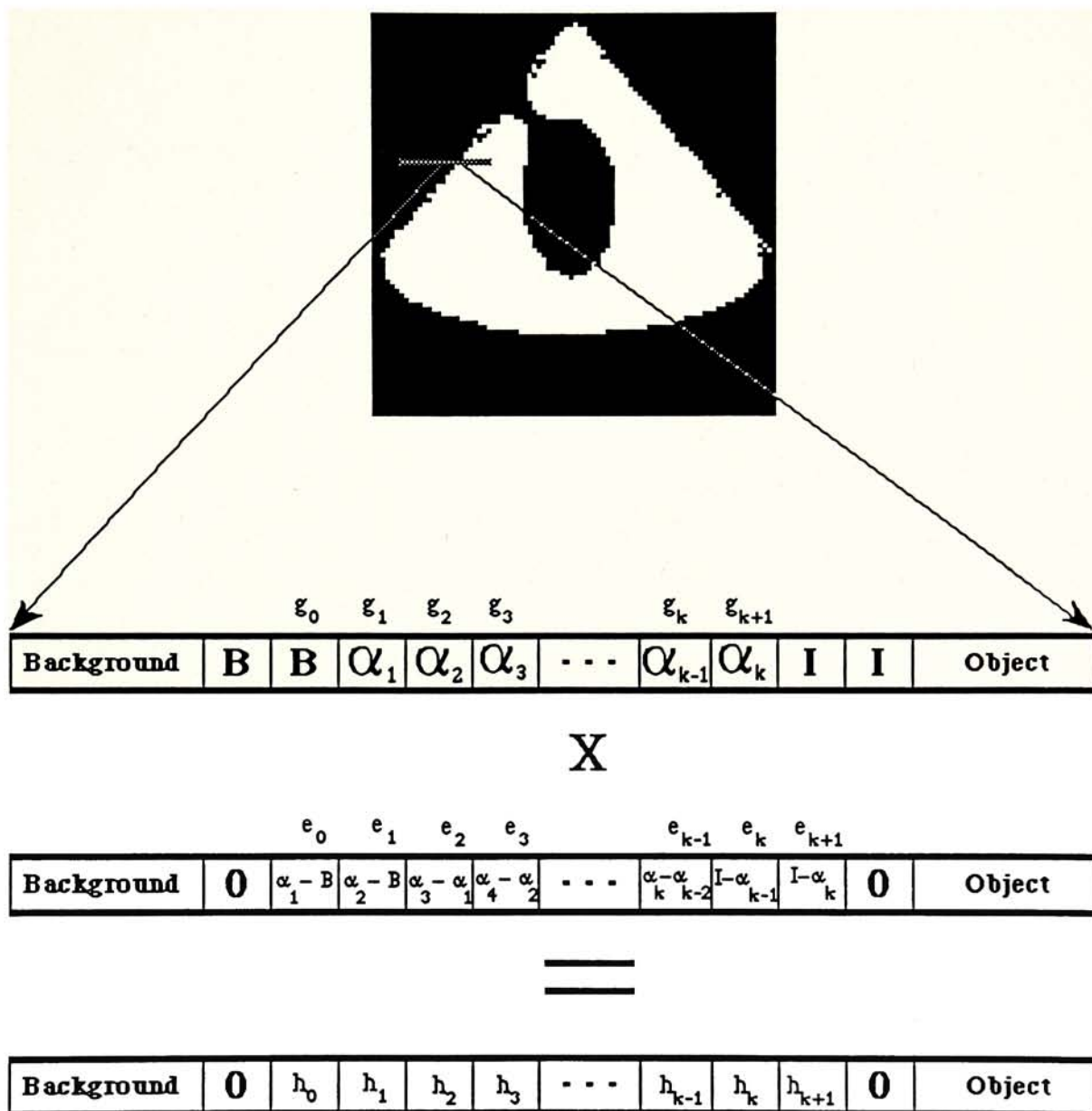


Figure 17 - Grey-grad values.

The product of the grey-level value, g_j , multiplied by the maximum derivative value, e_j , for each pixel, j , in the scan line can be represented by h_j . By taking the summation of h_j along the scan line near an edge transition, the following expression can be derived.

$$\sum_{j=0}^{k+1} h_j = \sum_{j=0}^{k+1} |e_j| g_j$$

The parameter k is the last pixel of the edge transition. This can be expanded to the following expression.

$$\sum_{j=0}^{k+1} |e_j| g_j = (\alpha_1 - B)B + \sum_{j=0}^k (\alpha_{j+1} - \alpha_{j-1})\alpha_j + (I - \alpha_k)I$$

This can be re-written as the following expansion of terms.

$$= (B\alpha_1 - B^2) + \left(\begin{array}{l} (\alpha_1\alpha_2 - \alpha_1B) + \\ (\alpha_2\alpha_3 - \alpha_2\alpha_1) + \\ (\alpha_3\alpha_4 - \alpha_3\alpha_2) + \\ \vdots \\ (\alpha_{k-1}\alpha_k - \alpha_{k-1}\alpha_{k-2}) + \\ (\alpha_k I - \alpha_k\alpha_{k-1}) \end{array} \right) + (I^2 - I\alpha_k)$$

A resulting simplification is given by the expression below.

$$= I^2 - B^2 = (I - B)(I + B) = C(I + B)$$

To summarize, the resulting expression for the grey-grad statistic for a scan line is given by the following relationship.

$$\sum_{j=1}^N |h_j| = C(I + B)n$$

Again, n represents the number of edges crossed by the scan line. This concept can be extended to the entire two-dimensional image given by the following expression.

$$\sum_{i=1}^N \sum_{j=1}^N |h_{ij}| = C(I + B) \sum_{i=1}^N n_i = C(I + B)m$$

In this expression, $N \times N$ is the dimension of the image in pixels, n_i represents the number of edge pixels in the i th scan line, and m represents the total number of edge pixels in the whole image. The working formula for the sum of the grey-grad statistic is expressed by the following equation.

$$\sum_{i=1}^N \sum_{j=1}^N |h_{ij}| = \sum_{i=1}^N \sum_{j=1}^N \max\{g_{ij}|e_{ij}|_x, g_{ij}|e_{ij}|_y\} = C(I + B)m$$

By incorporating the other edge contributions, the grey-grad statistic can be expressed by the following formula.

$$= \sum_{i=1}^N \sum_{j=1}^N \max\{g_{ij}|e_{ij}|_0, g_{ij}|e_{ij}|_{45^\circ}, g_{ij}|e_{ij}|_{90^\circ}, g_{ij}|e_{ij}|_{135^\circ}\} = C(I + B)m$$

The key expression for the automatic threshold selection can be established by combining the expressions for the sum of the grey-grad values and the sum of the grad values as a ratio to produce an elegant relationship to a threshold value as expressed below.

$$T = \frac{\sum_{i=0}^N \sum_{j=1}^N |h_{ij}|}{\sum_{i=0}^N \sum_{j=1}^N |e_{ij}|} = \frac{C(I + B)m}{2Cm} = \frac{I + B}{2}$$

The resulting grey-level threshold value, T , is a grey-level value between the grey-level of the background and the grey-level of the image. In terms of the image grey-level probability curve, this value represents the intermodal midway point in a histogram having a strictly bimodal distribution as shown in **Figure 13**.

The simplicity of the threshold selection process is made even more elegant by its implementation on the IP8500 system. The pixel-by-pixel differentiations and multiplications associated with the algorithm are amenable to the high speed architecture of the image processing system. The overall processing steps of this algorithm as implemented on the IP8500 can be summarized by the following diagram in **Figure 18** and outline steps.

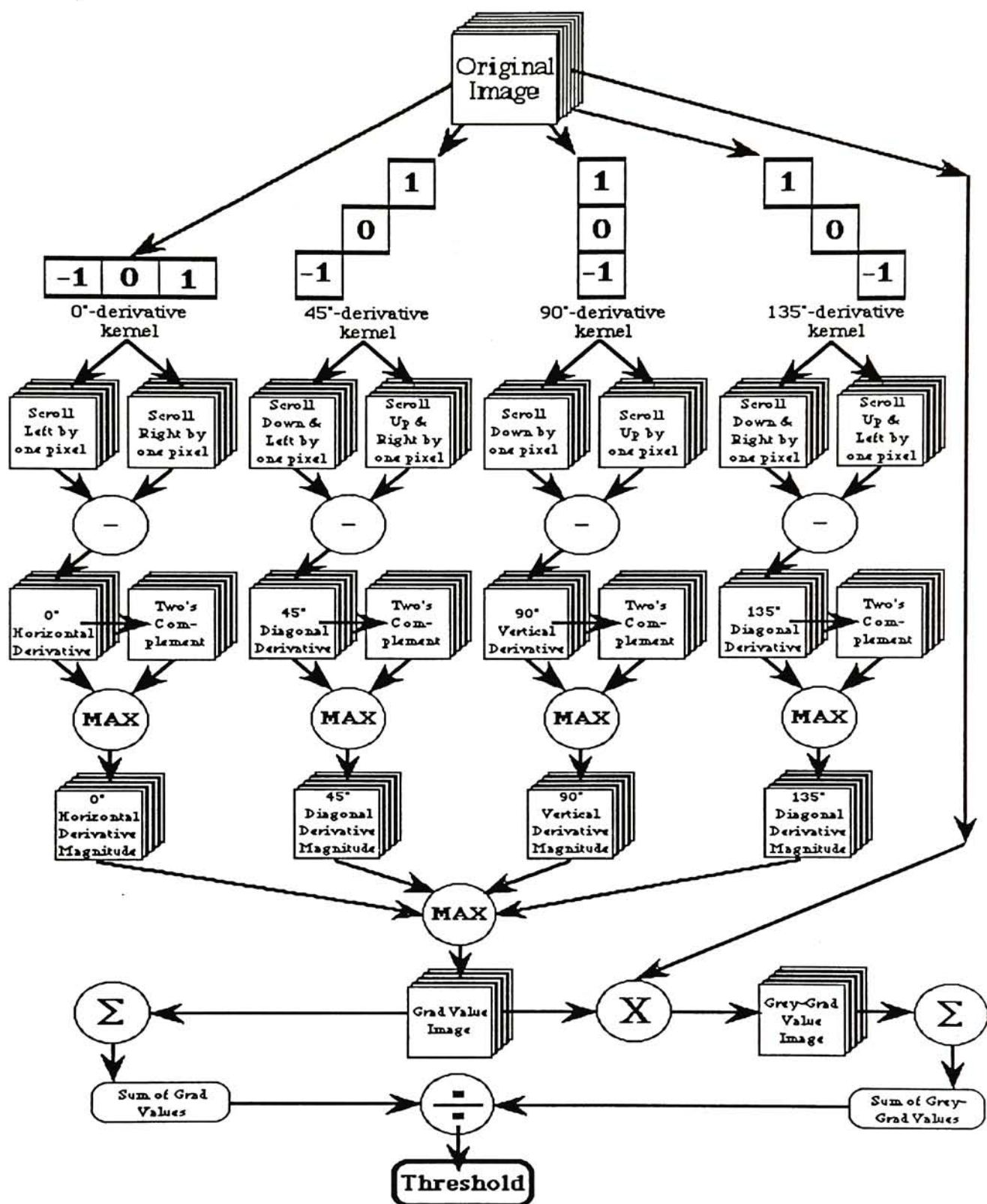


Figure 18 - Flow of RATS algorithm.

I. Calculation of the Sum of Grad-Values

A. Apply Derivative Kernel

- 1) Load original image into channel (0,1) as a 16-bit operand and zero channel (2,3)
- 2) Scroll channel (0,1) one pixel towards the desired orientation (0° , 45° , 90° , 135°), add channel (0,1) to channel (2,3) as a signed 16-bit operation placing the result into channel (2,3), and reset scroll.
- 3) Scroll channel (0,1) one pixel toward the opposite orientation (180° , 225° , 270° , 315°), subtract channel (0,1) from channel (2,3) as a signed 16-bit operation placing the result into channel (2,3), and reset scroll.
- 4) Save 16-bit result in channel (2,3). This represents the derivative of the original image for the desired orientation.
- 5) Repeat steps IA1-IA4 for each of the remaining orientations.

B. Magnitude (Absolute Value) of Derivatives

- 1) For a specific orientation, load the result of step IA4 into channel (2,3) and subtract the image from a zero constant placing the result back into channel (2,3). This will give a 16-bit two's complement image in channel (2,3).
- 2) Reload the saved result from Step IA4 into channel (0,1) and compare it with the two's complement result in channel (2,3) from step IB1 to create a 16-bit signed pixel-by-pixel maximum image in channel (2,3) using a DVP maximum operation..
- 3) Save 16-bit result in channel (2,3). This represents the absolute value of the horizontal derivative image.
- 4) Repeat steps IB1-IB3 for each of the remaining orientations.

C. Combine Magnitudes of Derivatives

- 1) For each orientation, load the magnitude of the derivative image for the desired orientation from step IB3 into channel (0,1).
- 2) Load the magnitude of the derivative image for the next orientation into channel (2,3).
- 3) Combine the two derivative magnitudes of different orientations into channel (2,3) by using the DVP 16-bit signed pixel-by-pixel maximum operator.
- 4) Load the next derivative magnitude of another orientation into channel (0,1).

Repeat step IC3-IC4 until all derivative magnitudes from the remaining orientations have been compared. Save the resulting grad image residing in channel (2,3).

5) Sum all the 16-bit values of the grad image in channel (2,3).

II. Calculation of the Sum of Grad-Values

- A. Load original image grey values as a 16-bit operand into channel (0,1).**
- B. Load image grad value as a 16-bit operand into channel (2,3).**
- C. Multiply channel (0,1) and channel (2,3) placing result into channel (2,3). This represents the 16-bit grey-grad image.**
- D. Sum all the 16-bit grey-grad values in channel (2,3).**

III. Calculate threshold value by dividing sum of the grey-grad values by the sum of the grad values.

It should be emphasized that the architectural features of the IP8500 enables expedient execution times of what normally would be a lengthy serial process. Most of the binary operations between two image operands usually require only a single frame time (1/30th second) for the IP8500 with more involved operations such as image multiplication requiring about three frame times (1/10th second). The bulk of the threshold selection calculation is also contained within the image processing system. This would not be true for many histogram based thresholding algorithms which may utilize the IP8500's fast histogram generator hardware, but require subsequent analysis of the histogram outside of the image processing system.

4.4 Quad-Tree/Multiresolution Data Structure:

A common data structure used in computer vision is the quad-tree (also known as a resolution pyramid) (Ballard 1982). This structure has the effect of quantizing images at progressively coarser resolutions from the original image. This operation starts with a full resolution image and progressively reduces the resolution by consolidating the grey-levels of four neighboring pixels into a single pixel value. This is repeated for all pixels in the image and has an overall effect of reducing the sampling resolution from 2^n to 2^{n-1} . For example, consider how a 32x32 pixel image can be recursively sub-divided into quadrants (**Figure 19**).

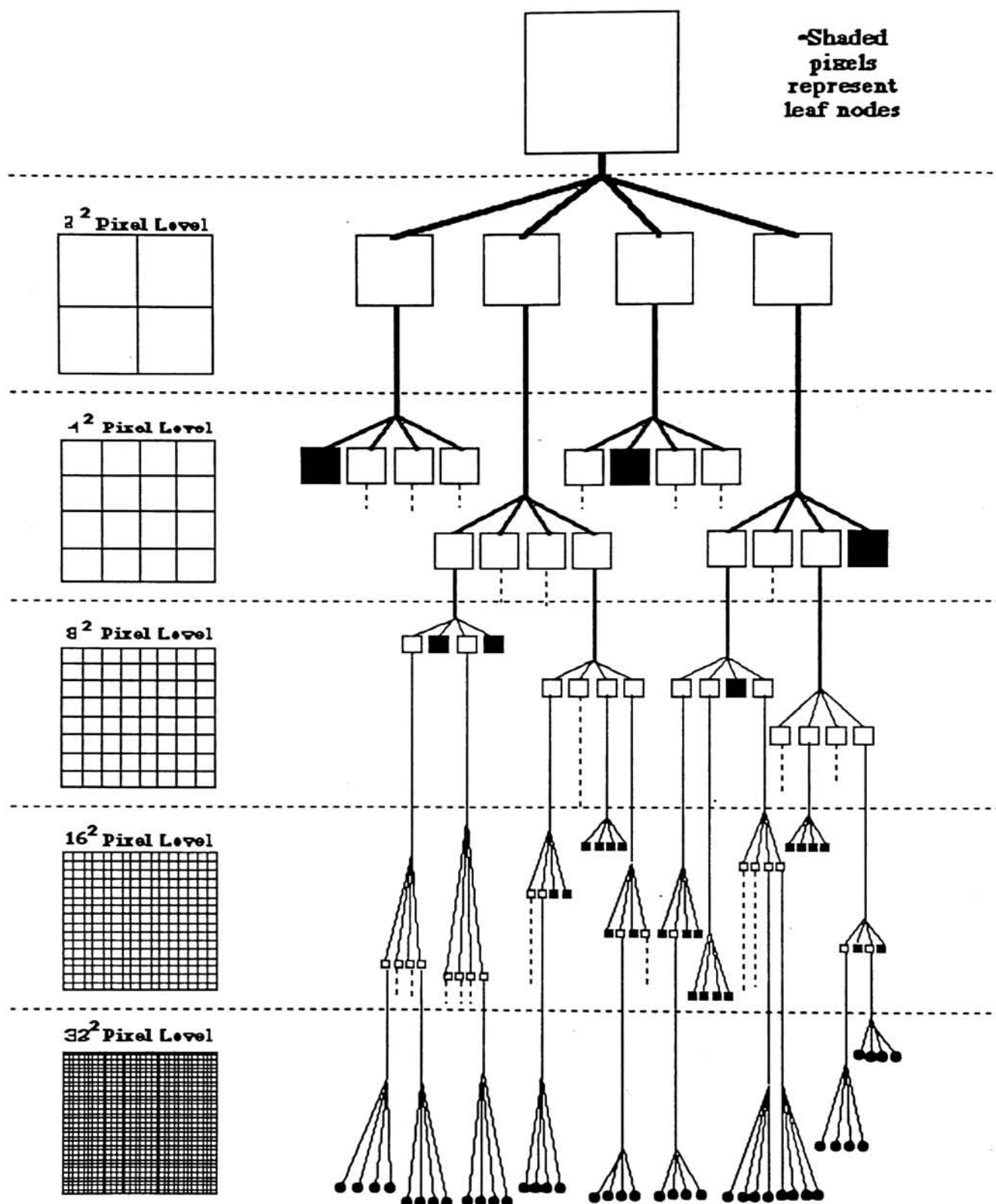


Figure 19 - Traditional Quad-Tree structure diagram.

To borrow a concept from imaging, the information contained at the top-most level node of the quad-tree would be analogous to the brightness information a single detector might observe for a given field-of-view. If that single detector were sub-divided into four separate detectors one-quarter the size of the previous detector, each would observe different brightnesses for each of the different quadrants. This would be analogous to the second level of the quad-tree. The lower levels of the quad-tree represent an increase in the number of detectors and a decrease in detector size with the leaf nodes of the tree at the lowest level representing individual pixels. Leaf nodes located at higher levels represent sections of uniform brightness which do not need to be represented by lower level nodes since doing so would be redundant. Although this process may seem initially disadvantageous and counterintuitive because of the reduction of information, it does have the advantage of reducing the dimensionality of the image by the systematic removal of image detail which might otherwise overwhelm automatic algorithms. This hierarchical reductionism of the image data allows a top-down approach to the search and processing of image information.

The classic implementation of the quad-tree structure involves the use of nodes and node-pointer data structures. The original design concept of the quad-tree for this project involved a self-referential C structure as shown below.

```

struct quad_node
{ short int x_coordinate;
  short int y_coordinate;
  short int quadrant_pixel_size;
  .
  .
  .
  additional node information
  .
  .
  .
  struct quad_node *quad[3];
};

```

The quad-tree node structure above can be categorized into four types of informational entries. The first type of information generally gives the x and y positions of the quadrant nodes in question. This allows any routines that refer to a particular quadrant node access to some form of positional information of the quadrant within the image; whether it be the center of the quadrant, lower left corner of the quadrant, upper right corner of the quadrant, etc. The

second piece of information is referred to as the quadrant pixel size. For each level of the quad-tree, each individual quadrant can be regarded as an aggregate parent pixel carrying some attribute or information about its offspring pixels found at the lower levels of the tree. These individual quadrant parent pixels, depending on their level in the tree, encompass a fixed area size of real image pixels, i.e., the pixels of the image at full resolution (leaf nodes in the tree). By maintaining this information in each node, it is possible to determine the fraction of the real image the node represents as well as the level the node occupies within the tree. The third type of information is the node pointers which link the parent nodes to the offspring quadrants. The last type of information can be any additional values pertinent to a specific problem which the node carries about its offspring nodes. An exemplary case of a quad-tree involving images might have each node contain the average grey-level value of its offspring nodes while it, along with its three other sibling quadrants, contributes to the average grey-level value of the parent node. There are other variations on this structure. The most common of these involve special threadings (node pointer links) which are used to optimize the traversal of the quad-trees.

The utility of a quad-tree in image processing is dictated by the ability to traverse the structure and extract information about the low-level high resolution nodes from the high level low resolution nodes. The quad-tree of an image as implemented above, does not provide lateral connectivity between sibling nodes short of traversing back up to the parent node and then down to the other sibling nodes. It is possible to provide interlinking threads between neighboring nodes, but complexity of traversal increases exponentially with connectivity. The tree traversal problem is an issue best addressed by sophisticated graph theory and search techniques which is beyond the scope of this study. A simplification of this data structure has been implemented to permit a solution to this constraint.

The assembly of a quad-tree can either use breadth-first or depth-first approach. Because of the self-referential structure of the quad-tree, a recursive algorithm provides an elegant method for creating the tree. Recursion allows designing, prototyping, and debugging of the procedures on much smaller $2^n \times 2^n$ pixel test windows. Recursive code is generally very compact and streamlined. An implementation of quad-tree creation other than a recursive procedure has proven to be very cumbersome. Details of the procedures and routines used in the creation of the quad-tree will be discussed in a later section dealing with a search algorithm used to determine positions of structuring element pixels in morphological operations. The two processes utilize routines that heavily exploit special IP8500 capabilities.

A variation on a theme of quad-trees is the basis of the multiresolution structure used in this thesis. A simplification of the quad-tree concept applied to images has been implemented after adopting some underlying assumptions about the images. The implementation also targeted the IP8500 architecture as a specification to which the data structure was designed. The major simplifying constraint assumes that the images used for the quad-tree structure are binary images. By making this assumption, a limit has been placed on the amount of information to be handled allowing a data structure to be implemented which partially addresses the problem

of quad-tree traversal. The binary image assumption allows each level of resolution to be represented by a $512 \times 512 \times 1$ bit image plane since only 1 bit per pixel is necessary to represent the on and off states. A quad-tree representation of a 512×512 pixel image would require ten levels of resolution starting from a 1×1 pixel level down to a 512×512 pixel level. If it is assumed that each resolution level requires a $512 \times 512 \times 1$ bit image plane, then ten 1 bit image planes are necessary to represent the entire quad-tree structure. At each resolution level, the on or off states of each pixel would be dictated by the presence or absence of on or off pixels at higher resolutions. By making the assumption that very little information is offered by the 1×1 pixel and 2×2 pixel images in the quad-tree, it is possible to limit the number of image planes to eight. This added constraint now allows the entire structure to be represented by a single memory channel in the IP8500 system. **Figure 20** illustrates this modified data structure.

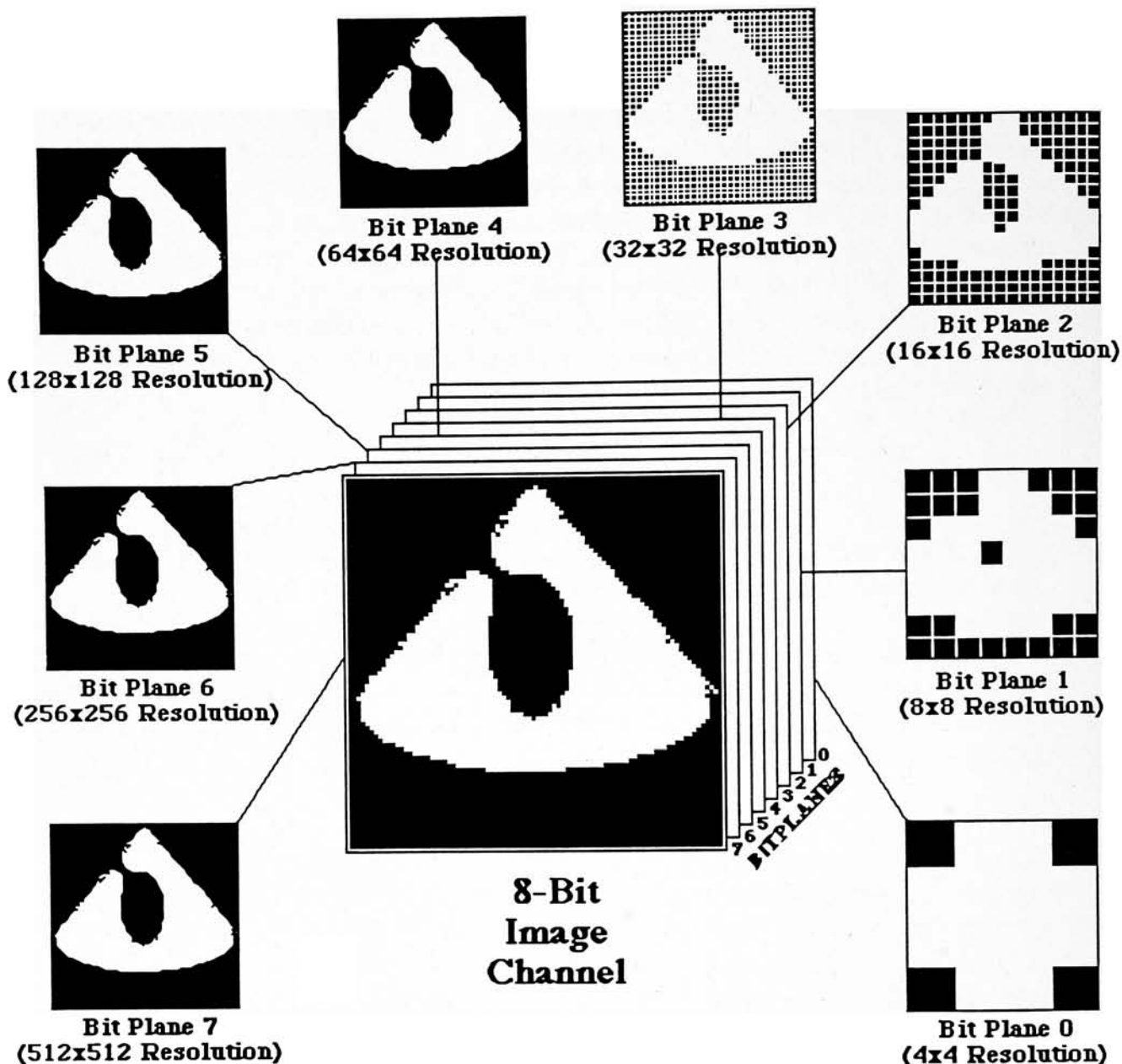


Figure 20 - Modified multiresolution representation.

The advantage of this data representation lies in the compact nature in which the multiresolution binary images are stored. What may have been an elaborate linked list structure has been reduced to a single 512x512x8-bit image. Specifying the proper resolution level becomes a matter of selecting the proper bit plane in the image. Addressing the pixels at a selected resolution is easily achieved since the pixel dimensions are pre-specified at each resolution level. The utility of this data structure in terms of the boundary detection problem will become apparent in the discussion of the following two algorithms.

4.5 Contour Following Algorithm:

The actual tracing of the boundary will be accomplished by a contour following algorithm (Pavlidis 1982). Assuming that the previous algorithms have generated a correct binary image free of salt-and-pepper noise, the contour following algorithm can automatically trace out the chamber-border. Because the image to be processed is a binary image, the two pixel classes are limited to ventricular wall pixels and cavity pixels. Any transition between the two classes of pixel indicate a boundary. This algorithm is the only routine in the set of selected procedures which does not heavily use the capabilities of the IP8500 system. The nature of the border search does not allow any form of parallelism in the algorithmic process to be implemented in the hardware of the image processing system. The algorithm itself is a very simple procedure involving a moving 3x3 pixel sample window shown **Figure 21**.

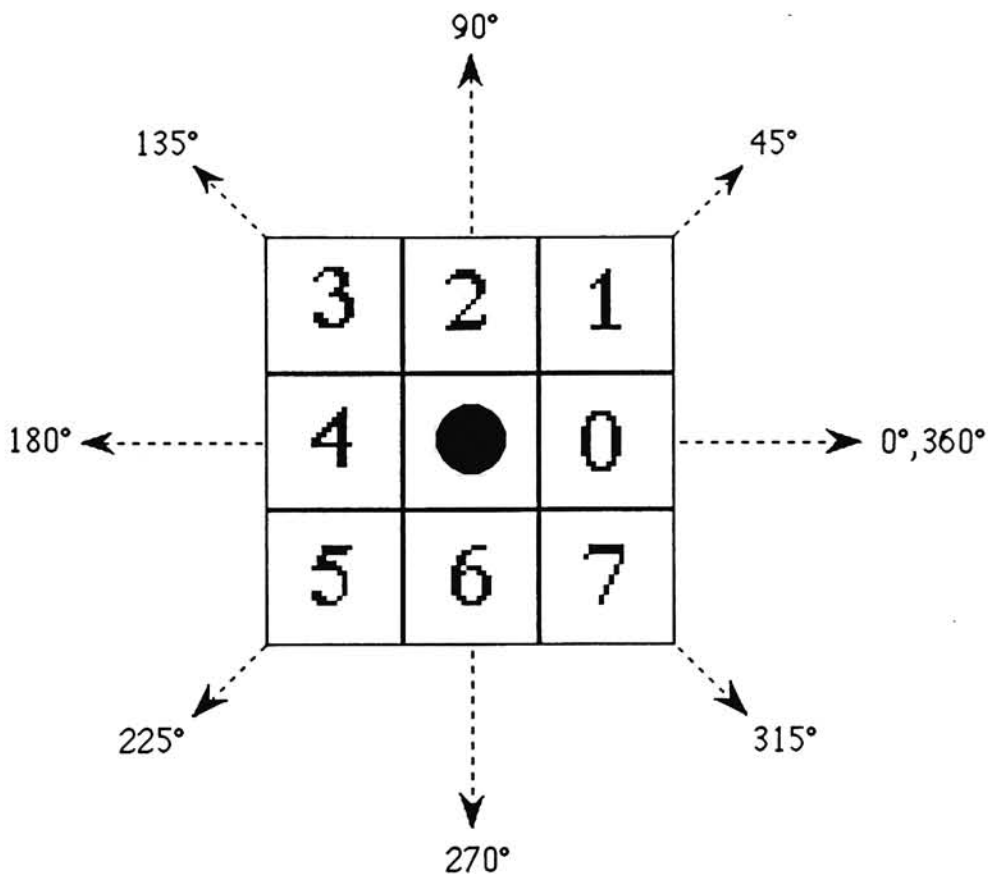


Figure 21 - Contour following kernel.

The window starts in the center of the image and scans across toward the left until a transition to a border pixel is detected. This initial transition represents the thresholded border between the chamber cavity and the ventricle wall giving an arbitrary first point of the border to start the

trace. Once this point has been established, the 3x3 window is placed over this point and a sweep of the neighboring pixels is made to determine the next boundary point. The neighboring pixels are sampled in a counter-clockwise order at 45° intervals (because of square tessellations) to determine the first pixel within the 3x3 window that matches the class of the center pixel, *i.e.*, a border pixel. This match indicates an adjacent edge pixel which allows the search window to be moved to that pixel where the process of sweeping across a new set of neighboring pixels is repeated. The starting search angle of the new window is set 225° clockwise of the very last search angle made in previous window position. This guarantees that any subsequent searches will not examine previously found border pixels. A record of each border pixel is kept as this process continues giving a list of vectors representing the boundary. **Figure 22** illustrates this process.

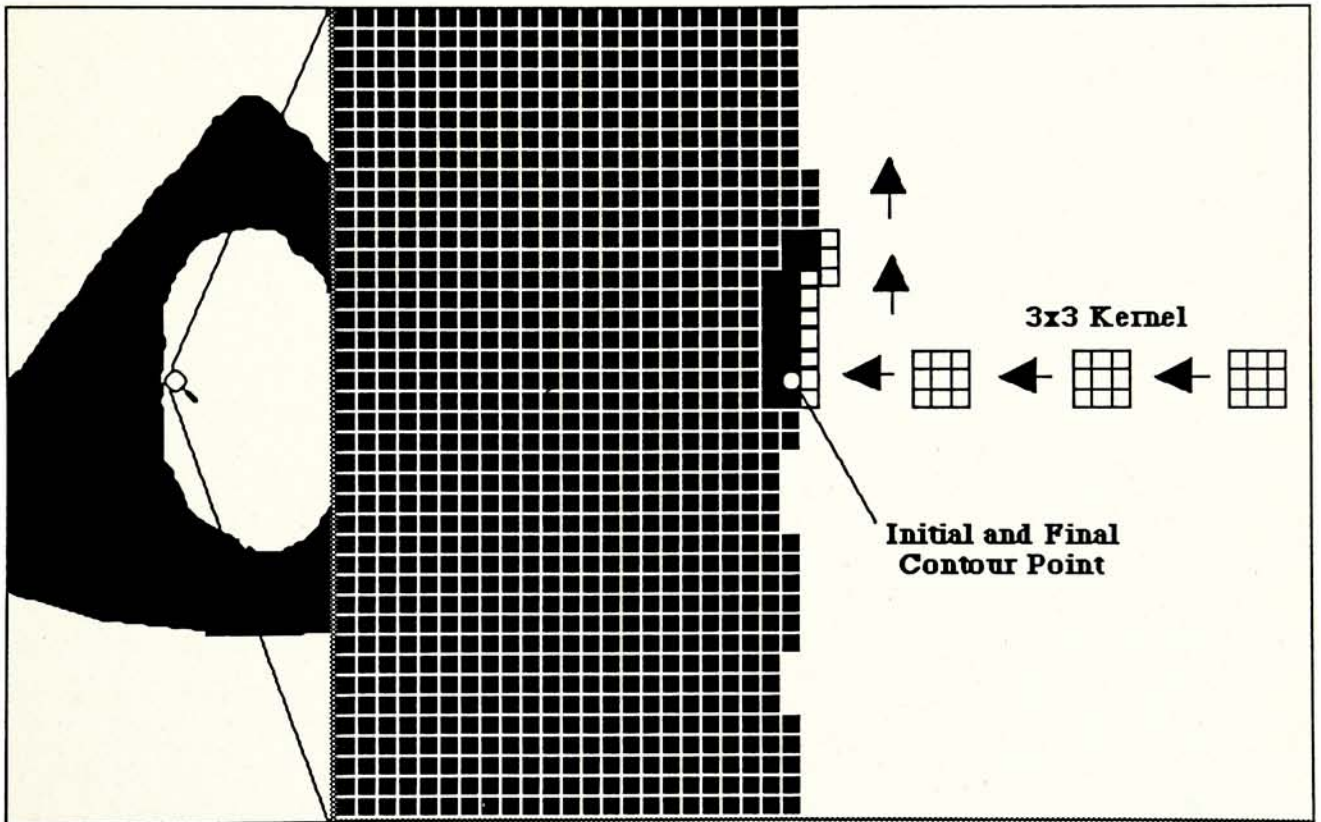
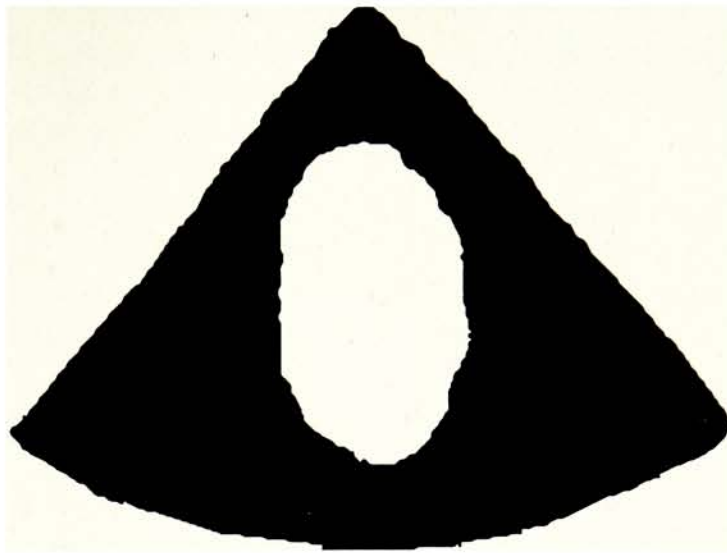


Figure 22 - Contour following.

The contour following algorithm will continue the border tracking process until the initial border pixel is encountered again. This condition signals a closed boundary. This condition, however, does not guarantee a closed chamber as shown in **Figure 23**.

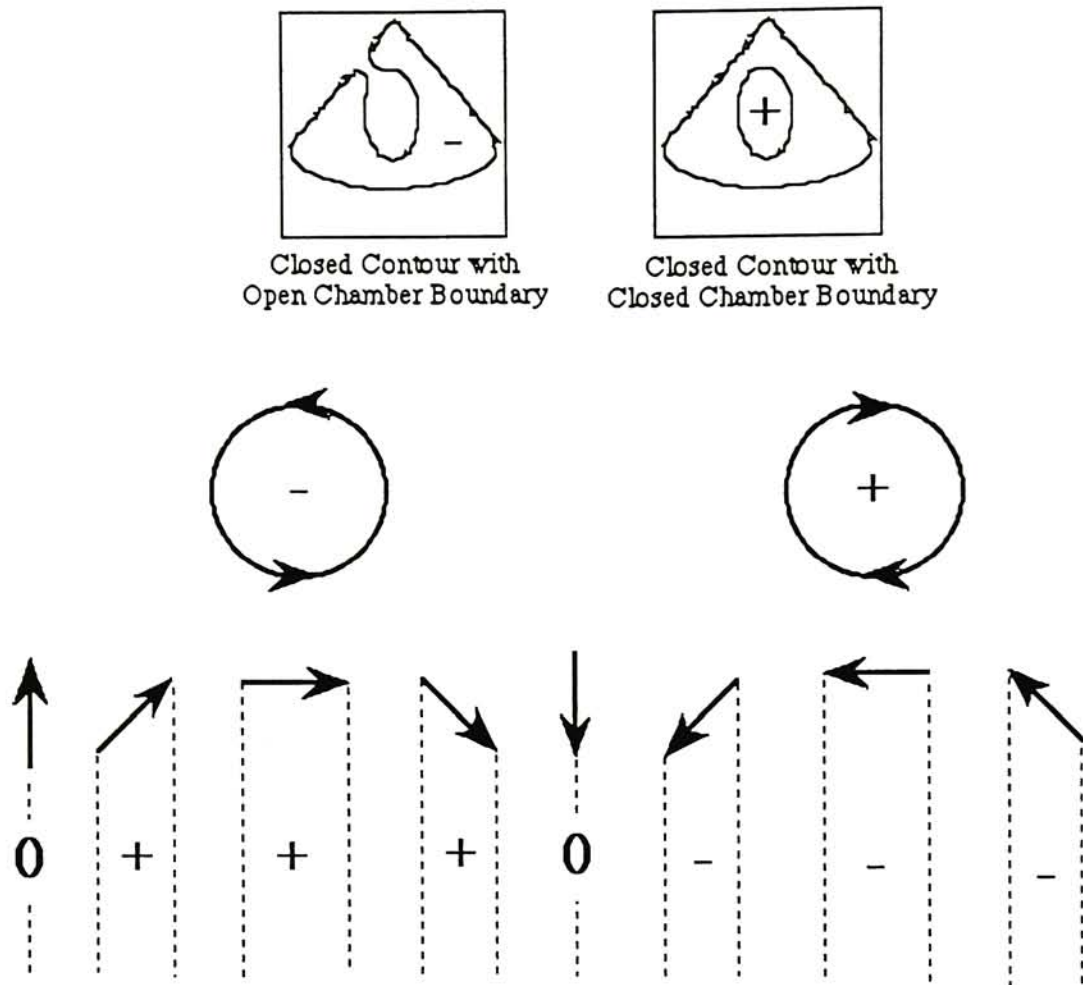


Figure 23 - Closed contour cases and corresponding area contribution convention of contours and vectors.

drawing. The left hand trace, however, represents a contour enclosing the boundary perimeter of strictly tissue area while the right hand trace represents a contour enclosing a cavity. An additional condition is, therefore, necessary to distinguish between a contour trace of the cavity and a contour trace of the tissues. This ability to discriminate between the two situations is based on establishing a definition of positive (cavity) and negative (tissue) bounded areas.

These areas can be computed by summing the area under each vector of the boundary and adopting the convention that all vectors having positive \hat{i} component contribute a positive area differential and all vectors having a negative \hat{i} component contribute a negative area differential as shown above.

A closed contour with an open chamber boundary will then generate a negative contour area whereas a closed contour with a closed chamber boundary will generate a positive contour area. This concept of using positive and negative areas is used frequently in mechanical engineering in the piecewise computation of area moments of inertia.

The ability to discriminate this condition is very important because the digitized echo images frequently contain dropouts in the heart wall due to the deterioration of the ultrasonic signal. This can also be complicated by the automatic threshold selection algorithm where the selection of a less than optimal threshold will tend to amplify this condition.

The next logical improvement to the automation stream would be to devise a way to measure the extent of the chamber opening and affect a closing process on the chamber boundary. Although the contour following algorithm can detect this condition, it cannot easily measure the gaps that frequently occur in the image. Some algorithms based on contour curvature metrics may be applied, but would involve probabilistic criteria which would again, as with the grey level threshold, require the selection of some form of confidence threshold level. Furthermore, multiple gaps in a single chamber image would render any mensuration algorithm based strictly on contour following, cumbersome and inefficient because of the accounting necessary to keep track of multiple contours and their attributes.

The resulting mensuration algorithm is a hybrid between the existing contour following algorithm and the quad-tree resolution representation of thresholded image. The contour following algorithm can be combined with the quad-tree data structure, described in the previous section, to determine a rough estimate of the size of the opening. This is based on the premise that the open boundary will at low resolutions of the quad-tree, become closed. By applying the contour following algorithm at several levels of resolution, it is possible to approximately measure the size the physical extent of the opening since this will correspond to the resolution size at which the opening is detected by the contour following algorithm. The subsequent closing operation of this condition will be addressed by the following section.

4.6 Morphological Operations:

The implemented solution to the open boundary problem entails morphological image operations. Morphological image processing differs from traditional linear system based image processing in that its approach to image analysis is based on geometric structures inherent in an image causing less distortion than normally observed with linear filters (Giardina 1988). Morphological operations utilize a structuring element defined in separate coordinate space to systematically alter the image in question. These structuring elements typically consist of simple shapes such as circular disks and square areas. Although the structuring elements can be of any shape and positioned anywhere in its own coordinate system, the structuring elements used will be limited to point symmetrical elements centered at the origin as a matter of

implementation simplicity. Conceptually, the process of these operations can be visualized as positioning a structuring element (e.g. a circle) along an image boundary and taking the traversed path of element to be the new image boundary to which the image will be expanded (i.e., dilated) or contracted (i.e., eroded).

A more formal treatment of morphological operations can be expressed in terms of Minkowski algebra. Dilation, for example, has the effect of expanding an image and can be expressed as a special case of Minkowski addition defined as

$$A \oplus B = \bigcup_{b \in B} A + b$$

where **A** represents a binary image to be processed defined in its own coordinate system, **B** represents a structuring element (essentially another binary image) defined in another coordinate system, and **b** represents all the points within **B**. Simply stated, the above set theory formalism specifies that the resulting dilated image is produced by translating image **A** by all the points **b** in **B** and taking the union of all resulting translations. Another way of representing this process involves defining the location of individual pixels in an image as the terminal point for a given vector from a defined image origin. This implies that an image becomes nothing more than a set of terminal points of a vector field emanating from the image origin. Likewise, a structuring element can be defined in the same manner with respect to its unique origin. By conceptualizing the image and structuring element as representations of vector fields, the Minkowski addition becomes the set of all points produced by the vector addition of each individual vector of the image vector field with each individual vector of the structuring element vector field as illustrated in **Figure 24**.

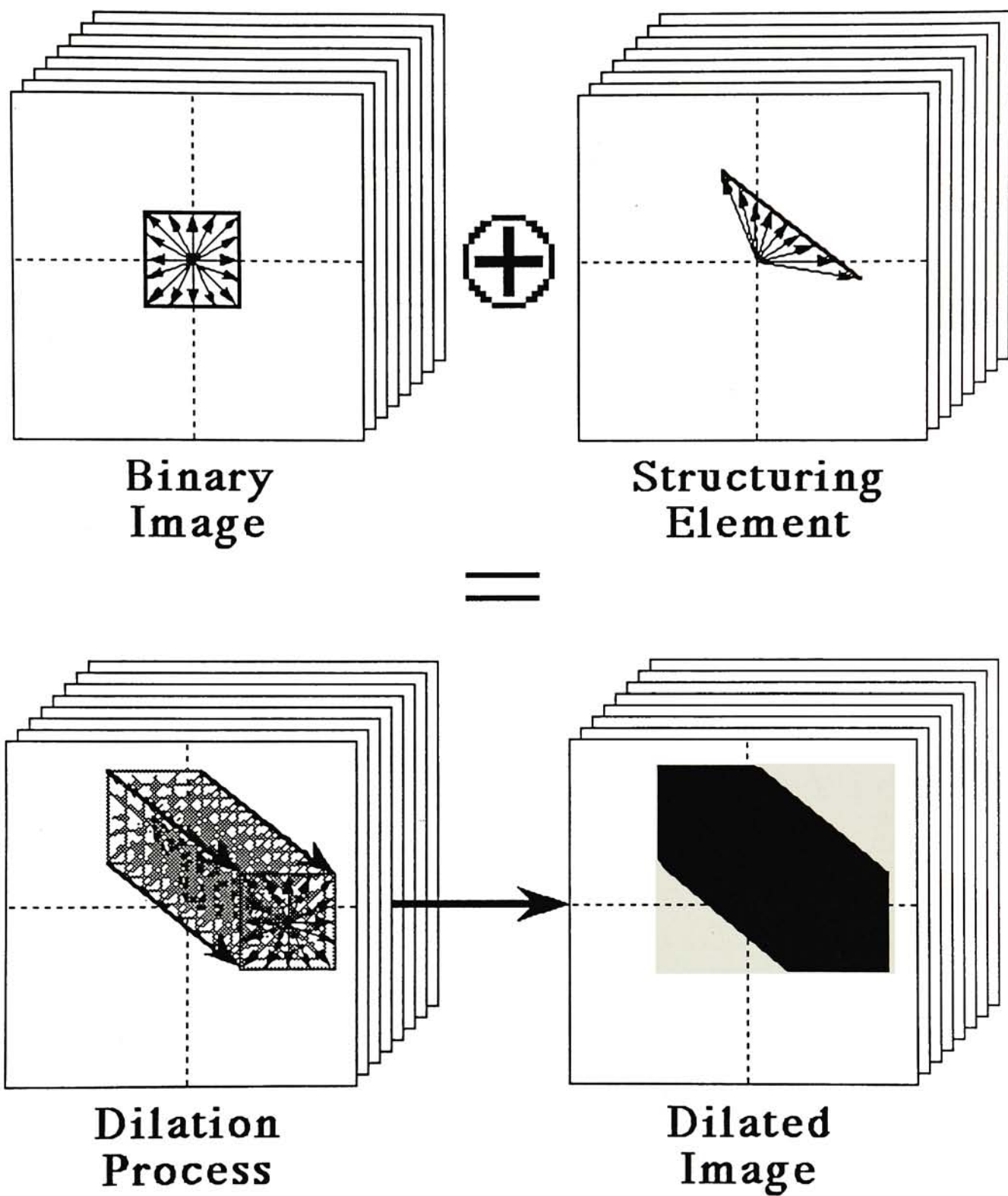


Figure 24 - Example of morphological operation (dilation).

. Either way, the dilation operation **D** is simply the Minkowski addition.

$$D(A, B) = A \oplus B$$

The contrary operation to dilation is erosion. Erosion has the opposite effect of "shrinking" the image and can be expressed as a special case of a Minkowski subtraction defined as

$$E(A, B) = A \ominus B$$

where

$$A \ominus B = \bigcap_{b \in B} A + b$$

The operation is similar to the Minkowski addition, except that the intersections of the image translations are taken instead of the unions. In terms of the vector field additions, erosion represents only those points which are common to all the vector additions.

Figure 25 diagrams how the morphological operations of opening and dilation can be applied to produce a closed contour to affect a proper boundary trace.

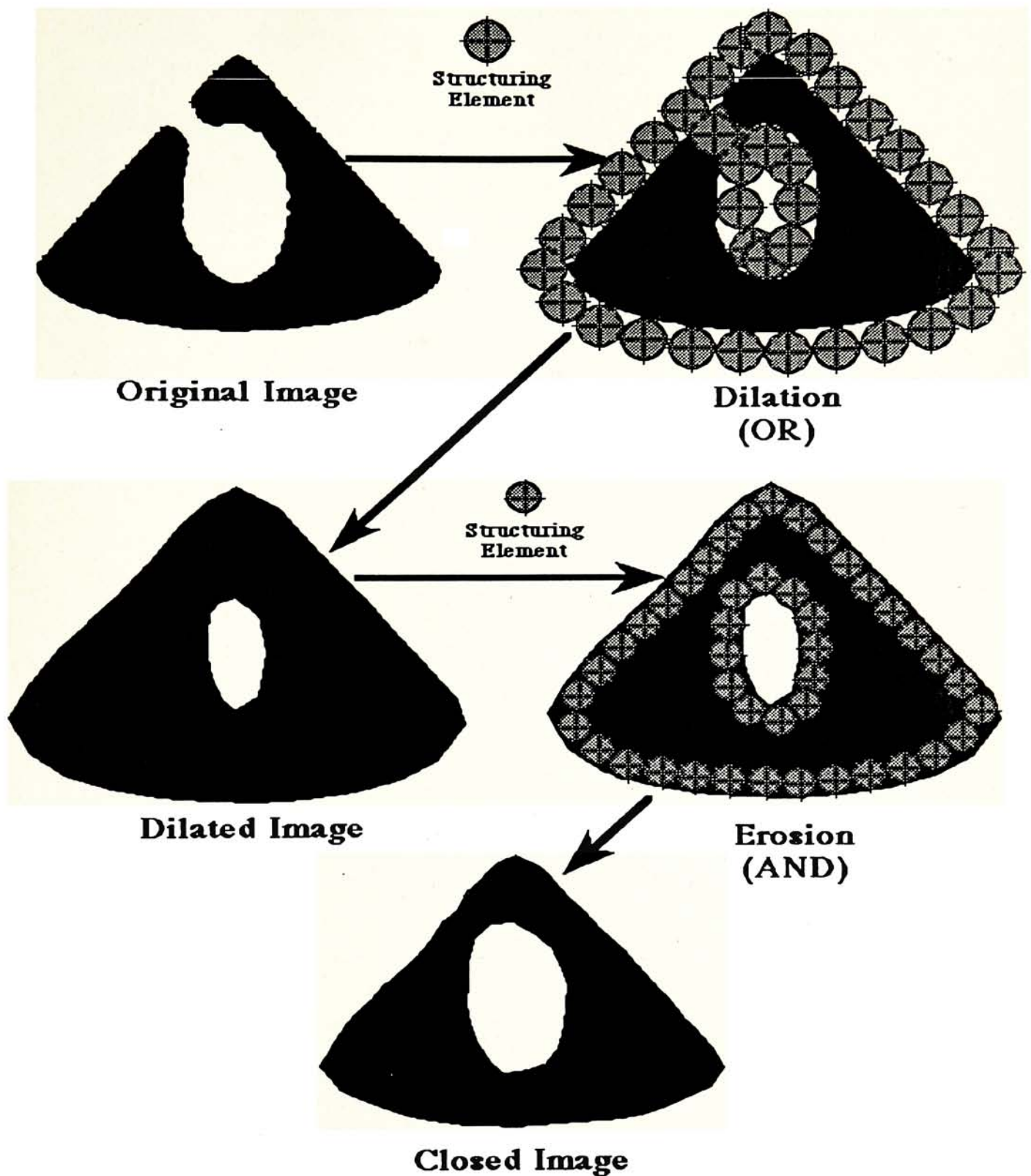


Figure 25 - Sample morphological closing sequence.

The utility of the closing operation is an obvious one. A major factor that has not been determined and is a persistent issue in morphological image processing involves the determination of the proper shape and size of the structuring element. As stated previously, the choice of a structuring element for this specific application has been limited to a point symmetric structuring element, namely a circular disk centered at the origin. Although this may not be the optimum structuring element shape for this application, it does offer an intuitive closing effect for this specific set of images. Selection of the optimum structuring element shape is best treated as a separate study in morphological image processing. Since the structuring element shape has been established, the size and sequence of morphological operations remain the other variables to be determined. This procedure will be detailed in the following section dealing with algorithm integration.

The implementation of the morphological operations can be decomposed into primitive operations inherent to the IP8500 processing repertoire. These include the basic image scrolls and image bitwise logical operations performed by the DVP. Because of DVP limitations, its use implicitly limits the image and structuring element dimensions to 512 x 512 pixels. The relevant structuring elements are origin centered which makes the rotation primitive unnecessary due to symmetry. Actual execution of the morphological operations using the IP8500 commences with the loading of the image into a spatially translatable operand channel and a stationary result channel in the image processing system. The structuring element is loaded into its own memory channel where it is used to control the scrolling of the operand channel. A search is made on the channel containing the structuring element to determine the positions of each individual pixel. As each pixel of the structuring element is located, its position is noted. The operand channel is then translated by the vector connecting the origin of the structuring element channel to the structuring element pixel. The appropriate binary boolean operator is then applied to the scrolled operand channel and the stationary result channel. The intermediate image is left in the result channel. **Figure 26** illustrates this process for one structuring element pixel.

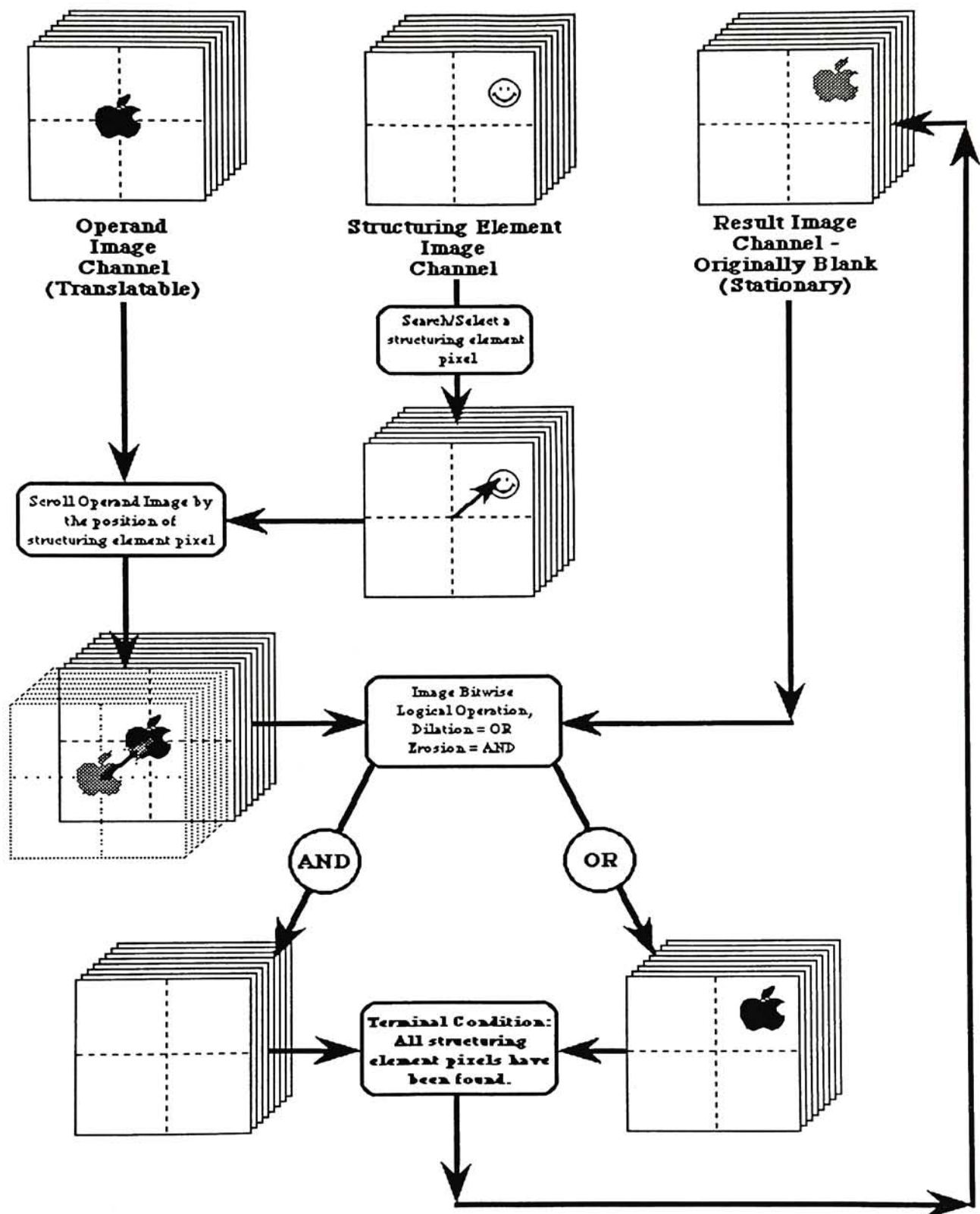
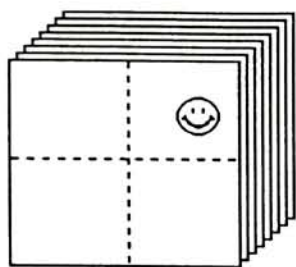


Figure 26 - DVP implementation of morphological operations.

This process is repeated on the intermediate image until all pixels of the structuring element have been searched and the corresponding translation and boolean operation applied.

Early implementations of the structuring element pixel search resorted to a simple search algorithm in which all the pixels in the structuring element memory channel was interrogated sequentially by column and then by row. Although simple to implement, this type of search algorithm does not utilize any of the sophisticated capabilities of the IP8500 and was destined to search every pixel in the 512 x 512 memory channel space. A later improvement in the search algorithm utilized the same techniques and procedures used in building the quad-tree data structure mentioned previously. This allowed existing code to be applied to a different application with minor modification. This quad-tree type search follows a similar strategy found in a one-dimensional array binary search. A binary search employs a divide and conquer technique to recursively narrow the search space until the desired element is encountered. The heart of this quad-tree type search lies in the exploitation of the fast-histogram board present with the DVP hardware. Although primarily used for the calculation of grey-level probability distributions, the fast-histogram board provided an expedient means to test the presence of pixel in a given region of interest. The region of interests begin as large low resolution quadrants. These are then tested for the presence of structuring element pixels. Absence of structuring element pixels in a given quadrant, as determined by the histogram board calculated statistics, makes further detailed search in that particular quadrant unnecessary. The presence of pixels, on the other hand, would cause the search to continue by recursively subdividing the quadrant into smaller regions until a homogeneous quadrant of structuring elements is encountered. **Figure 27** shows the subdivision partitions that leads to the location of a homogeneous square subimage. The homogeneous square subimage dimension range in the order of a single pixel to 256x256 pixels. In cases where the quadrant size is greater than one pixel and contains all structuring elements, the search for the remaining pixels in the quadrant reverts to a sequential type search.



**Quad-Tree type search for a
single structuring element pixel
(*e.g.* right eye of smiley face)**

**Structuring Element
Image
Channel**

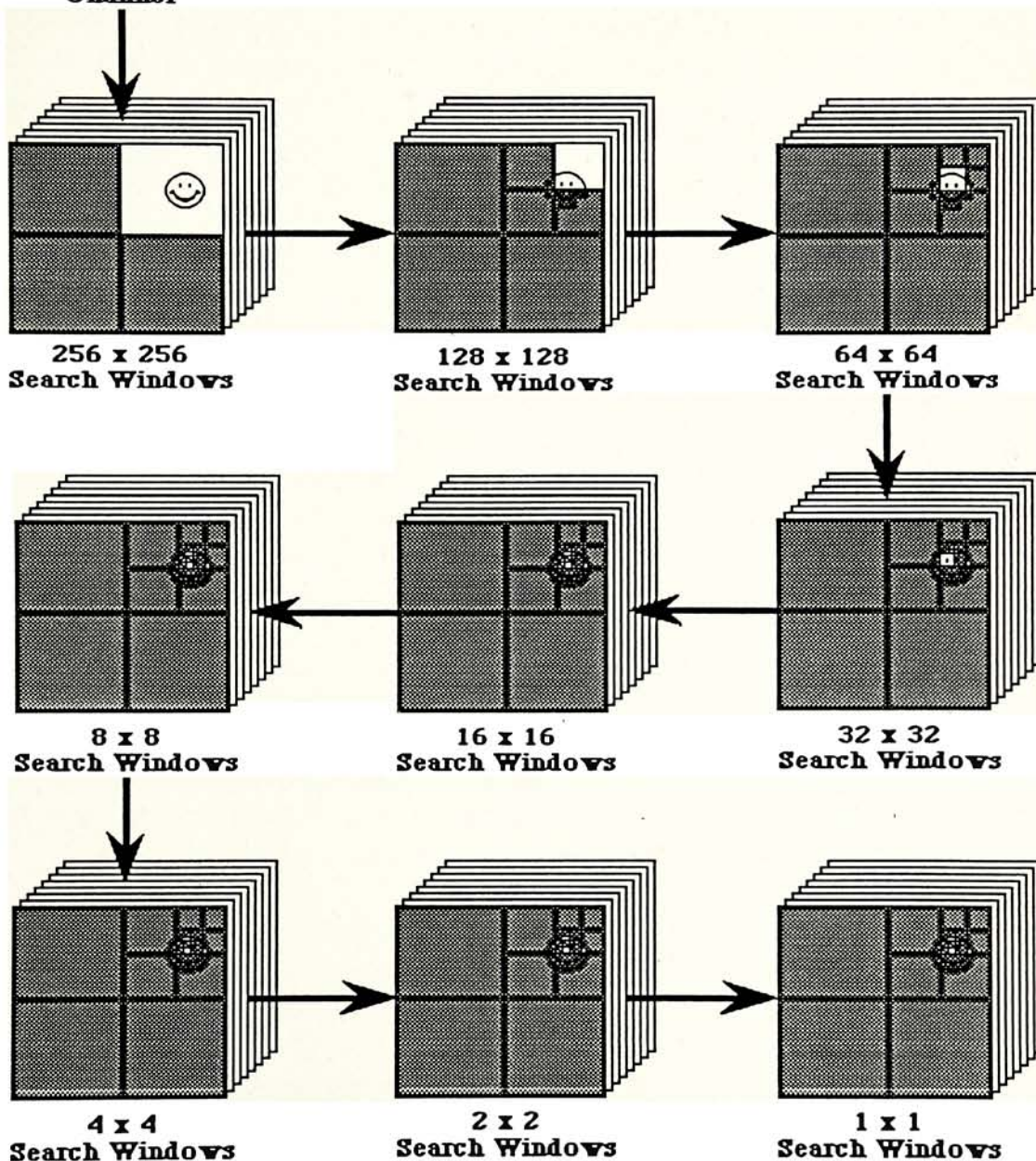


Figure 27 - Quad-Tree type subdivision search.

4.7 Algorithm Integration:

The previous sections have described individual algorithms in detail without any specifics as to how they interact. Aside from the pre-processing algorithms which are stand alone processes, the quad-tree representation, the contour following algorithm, and the morphological operation have been integrated to form an adaptive solution to the open boundary problem.

Recall that after preprocessing and thresholding the resulting image is a binary representation of a classification between tissue and background/cavity. In an ideal situation, the binary image should be sufficiently thresholded to allow the left-ventricle chamber to be automatically traced by the contour following algorithm. As mentioned previously, this ideal situation is more of an exception rather than the rule as most of the the binary images exhibit boundary dropouts. The solution chosen to cope with this condition involves the transformation of the binary image into the quad-tree resolution form. This is the data structure discussed in the quad-tree section where a single image plane virtually represents multiple resolution of the binary image in several bit planes. Once in this format, the contour following algorithm is applied to find the resolution at which dropout details are lost. the high resolution bit plane of the image and the resulting area enclosed is measured. If the contour generated is not of the left ventricle cavity, a negative area value will be computed. This will signal the contour following algorithm to proceed to the next bit plane containing the lower resolution representation of the binary image. This process is repeated until the resolution bit plane generating a positive area (*i.e.* a closed contour of the ventricle cavity) is found. The resolution of this bit plane is noted as well as the size of the super-pixel (in terms of full resolution pixels). For example, **Figure 20** shows a diagram of a binary image in the multiresolution form of a sample echo image with a dropout. If the contour following algorithm were applied as described above, the contour areas calculated from bit planes 7 to 3 would have generated negative values. Bit plane 2, however, would have generated a positive value signalling a correctly closed contour. Bit plane 2 is an image with an effective resolution of 16x16 super-pixels. This translates to each super-pixel having a size 64x64 full resolution pixels. This indicates that the size of the dropout gap is approximately 64 full resolution pixels. This information now gives the morphological algorithms a starting value with which to initiate a closing.

The resolution value and super-pixel size generated by the contour following algorithm can now be used as a guide to choose the structuring element for the morphological operations. In this specific example, a circular structuring element centered at the origin with a diameter of $64 \cdot \sqrt{2}$ would be used. The factor of $\sqrt{2}$ is used to take into account the diagonal dimension of the square tessellate pixels. The image channel is then dilated by this structuring element and then eroded using the same type of structuring element, but with half the diameter as shown in **Figure 25**. An erosion with the same size structuring element as the one used for the dilation would yield a smooth image but still with an open cavity. By choosing a structuring element half the size of the one used for the dilation, it is possible to erode the image and still maintain a

closed image. Since the sizes of the two structuring elements are known, a correction factor equalling half the difference of the diameters of the dilation structuring element and the erosion structuring element can be used to correct the contour trace made on the closed image. This correction factor equals the resultant dimension by which the image was "thickened" by the morphological operations. This correction is applied to the final trace by extending the trace point distances from the centroid by the amount of the correction factor. The centroid is simply given as (\bar{x}, \bar{y}) and is calculated when the image has been successfully closed.

Because of the bit plane structure of the multi-resolution representation of the binary image, all of the bit planes are subjected to the same morphological processing effect. This means that any closing effects applied to bit plane 7 (**Figure 20**) will be applied to all bit planes. This feature is useful because in the event that the image is not closed in the first pass of morphological operations, the subsequent application of the contour following process will choose a new structuring element size based on the processed bit planes instead of the original bit planes of the multiresolution image. Processing will continue until the image has been closed.

5. Results:

Six frames of images were chosen from a single cardiac cycle as test images. They were chosen from the same cardiac cycle to minimize the possibility of variation in image gain and bias caused by the equipment or affected by the sonographer. The six images are illustrated below in **Figure 28** in masked form using the sector mask described in the pre-processing section.



Figure 28 - Original images (masked).

The six images were designated with the identifying code of D180F(Frame number). Since the images were acquired at video rates, the frame number indicates the position of the image within a 30 frame video cycle. **Figure 29** shows the resulting images after a spatial convolution with a 15x15 Gaussian convolution kernel.

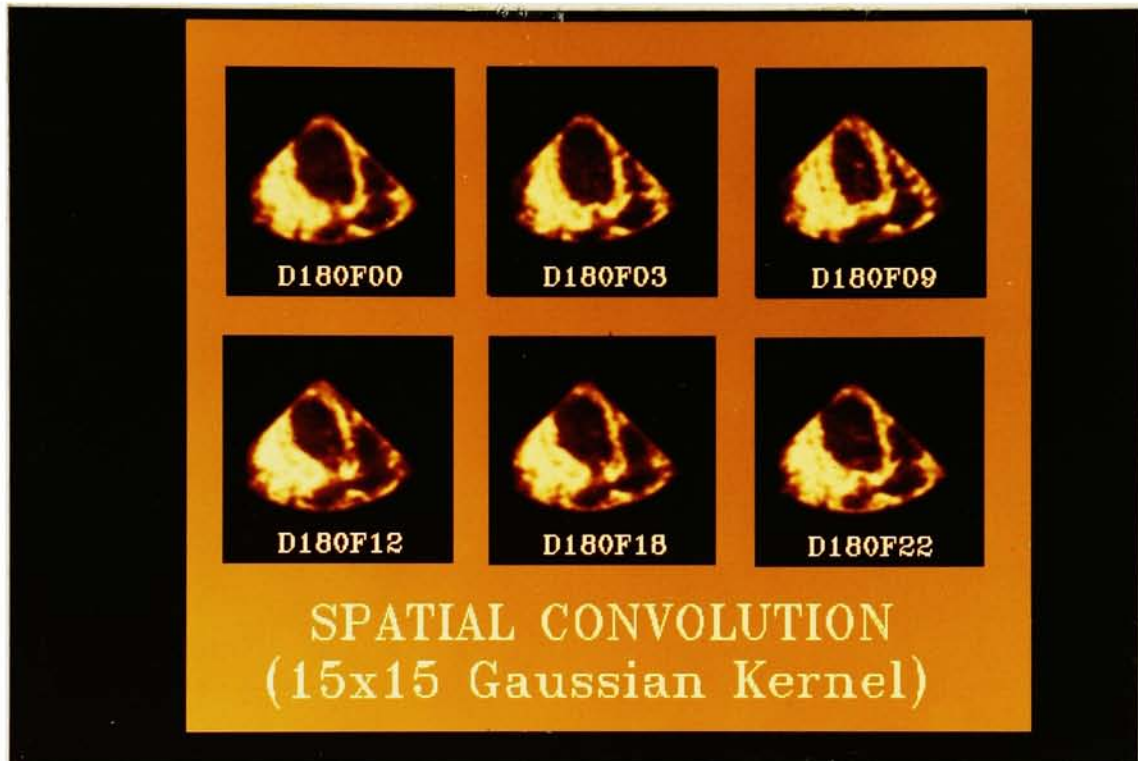


Figure 29 - Smoothed images.

The images were then thresholded, as shown in **Figure 30**, using the values selected by the RATS algorithm.

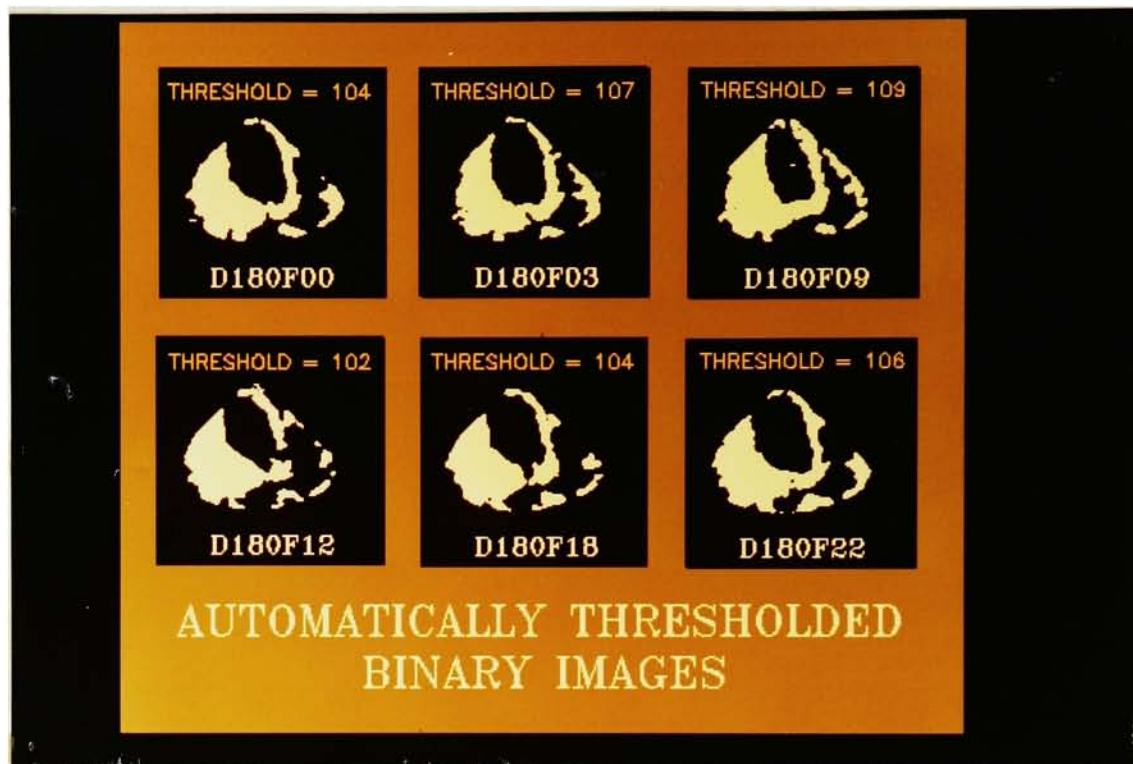


Figure 30 - Thresholded Images.

These values are compared with the thresholds selected interactively in **Table 1**.

Table 1 - Threshold Statistics

Image	Computer Threshold	Manual Threshold	%Error [grey levels]
D180F00	104	84	23.81
D180F03	107	73	46.58
D180F09	109	84	29.76
D180F12	102	85	20.00
D180F18	104	83	25.30
D180F22	106	95	11.58
Standard Deviation	2.50	6.99	22.48 RMS

The corresponding plot of computer vs. manual threshold values is shown in **Figure 31** along with error bars .

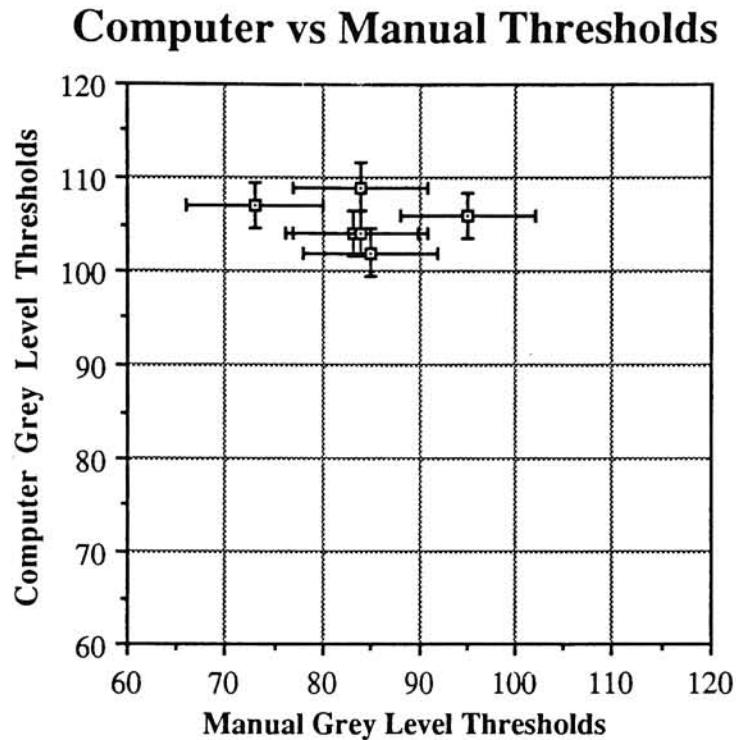


Figure 31 - Threshold plot.

Since there are only 255 discrete values in the grey scale of the images, errors translate into very significant changes in the resulting binary image. Although the errors of the computer generated thresholds are relatively large when compared to the interactively determined thresholds, their effect on the overall result is masked by the application of the morphological operators. The variance of the thresholds selected by the automatic method for a set of similar images is much smaller than those chosen interactively. This is supported by the error bars plotted in graph.

As described in the previous section, the resulting binary images are transformed into a quad-tree resolution image as shown below in **Figure 32** in a manner where the different resolution image bit planes have been visually enhanced.

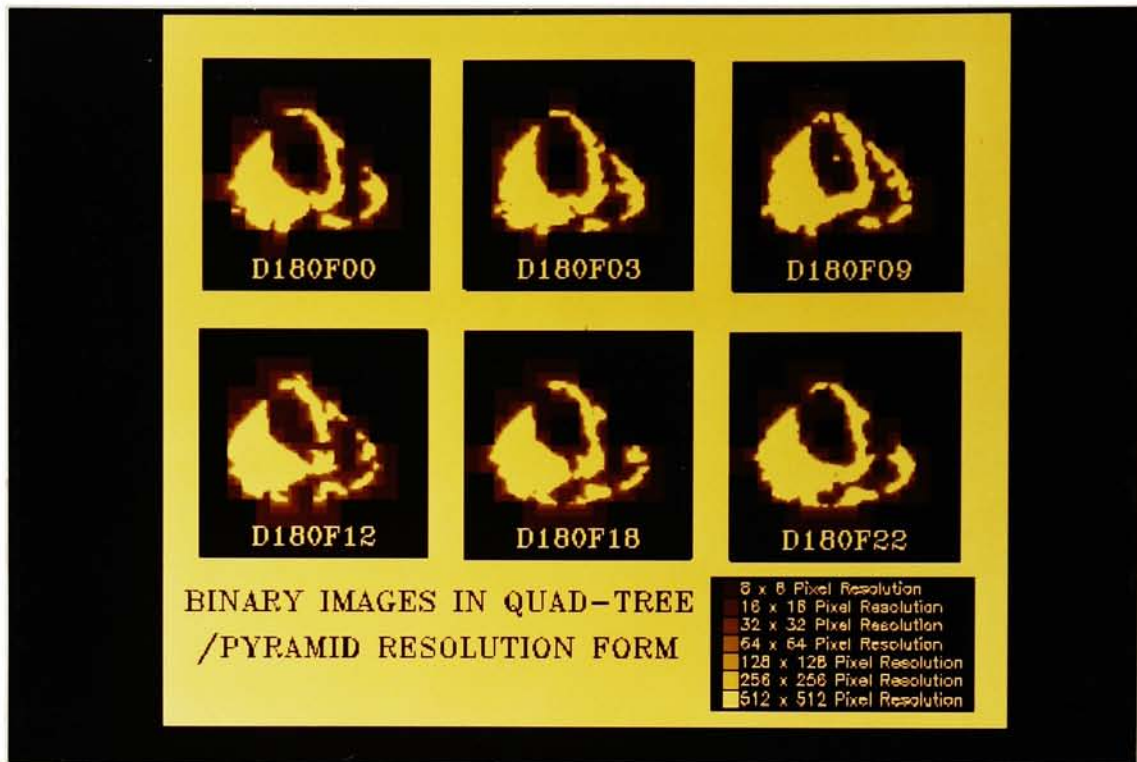


Figure 32 - Multiresolution images.

The images are then subjected to a dilation/erosion sequence to affect a closing illustrated below in **Figure 33**.

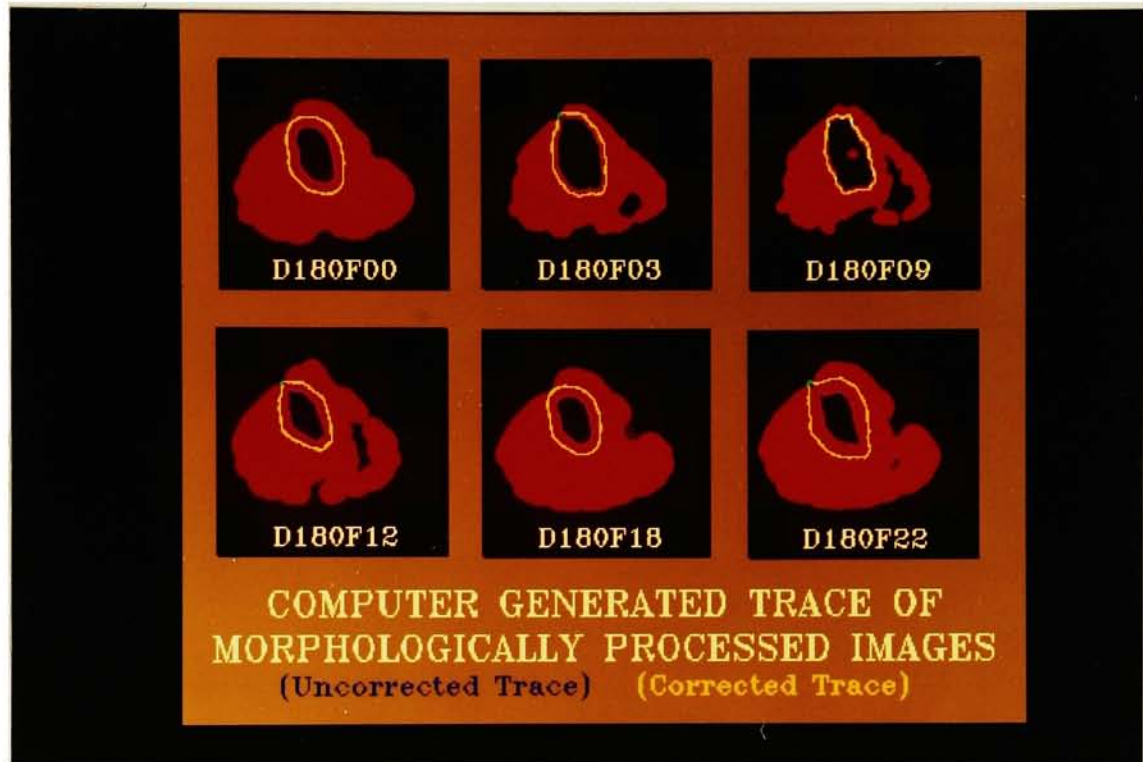


Figure 33 - Morphologically processed images with traces.

This dilation/erosion process is adaptively repeated until the image is closed. This iterative sequence is summarized in **Table 2**.

Table 2 - Morphological Processing Statistics

Image	(Dilation/ Erosion) #	Correction Factor [Pixels]	(D/E) Circle Diameter [Pixels]
D180F00	1	23	(91/45)
D180F03	1	11	(45/23)
D180F09	1	6	(23/11)
D180F12	7	21	(45/23),(11/6),5*(6/3)
D180F18	1	23	(91/45)
D180F22	9	30	(45/23),(23/11),3*(11/6),4*(6/3)

This table gives the number of dilation/erosion sequences needed to close each image along with the resultant correction factor reflecting the net width by which the image was "thickened". The final column lists the dilation/erosion diameters of the circular structuring elements and the sequence in which they were applied.

The resulting corrected computer contour traces were then visually compared by overlaying them over the manual traces and the original image as shown below in **Figure 34**.

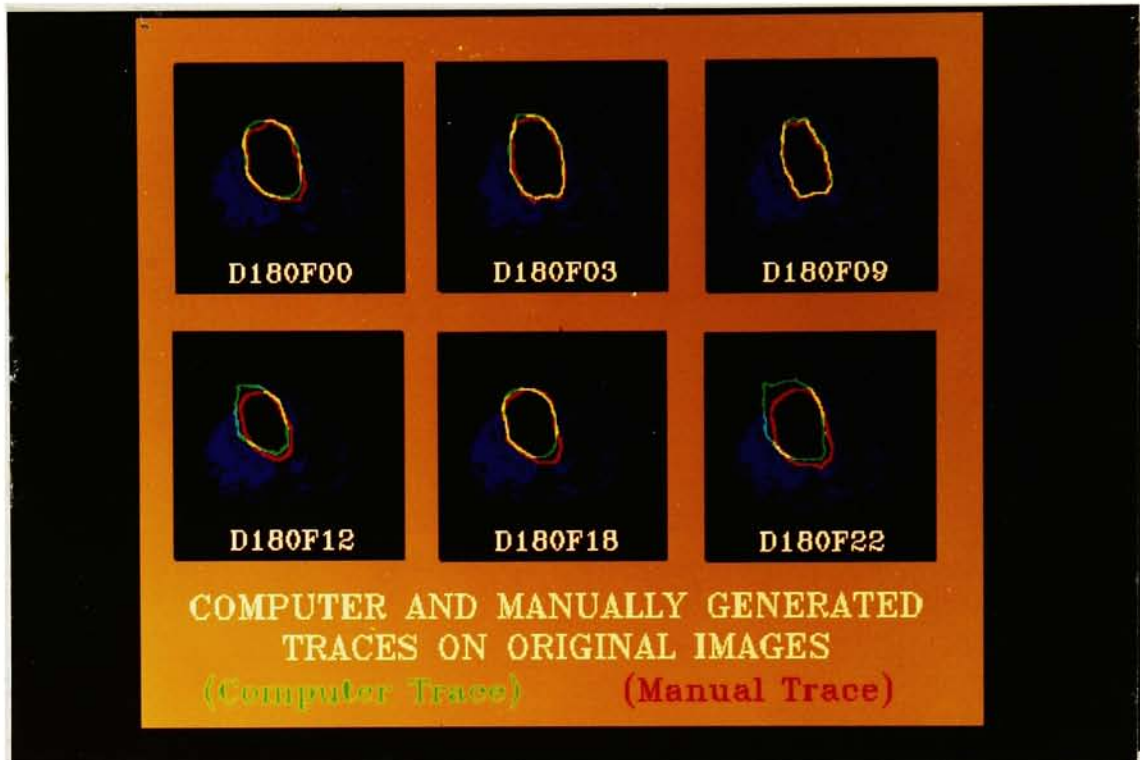


Figure 34 - Computer and Manual Traces on Original Images.

A more quantitative comparison was made by calculating the root mean square (RMS) error of the contour distance of the computer generated trace and the manual generated trace to the centroid point of the manual trace. The contour trace distances to a given centroid can be plotted as a function of angle to yield a one-dimensional relationship in rectangular coordinate space as shown in **Figures 35a-b**. These plots give a graphical comparison of the computer vs. manual generated traces illustrating the trends of the traces and locations of large discrepancies.

Contour Distance from Centroid (Manual)- D180F00

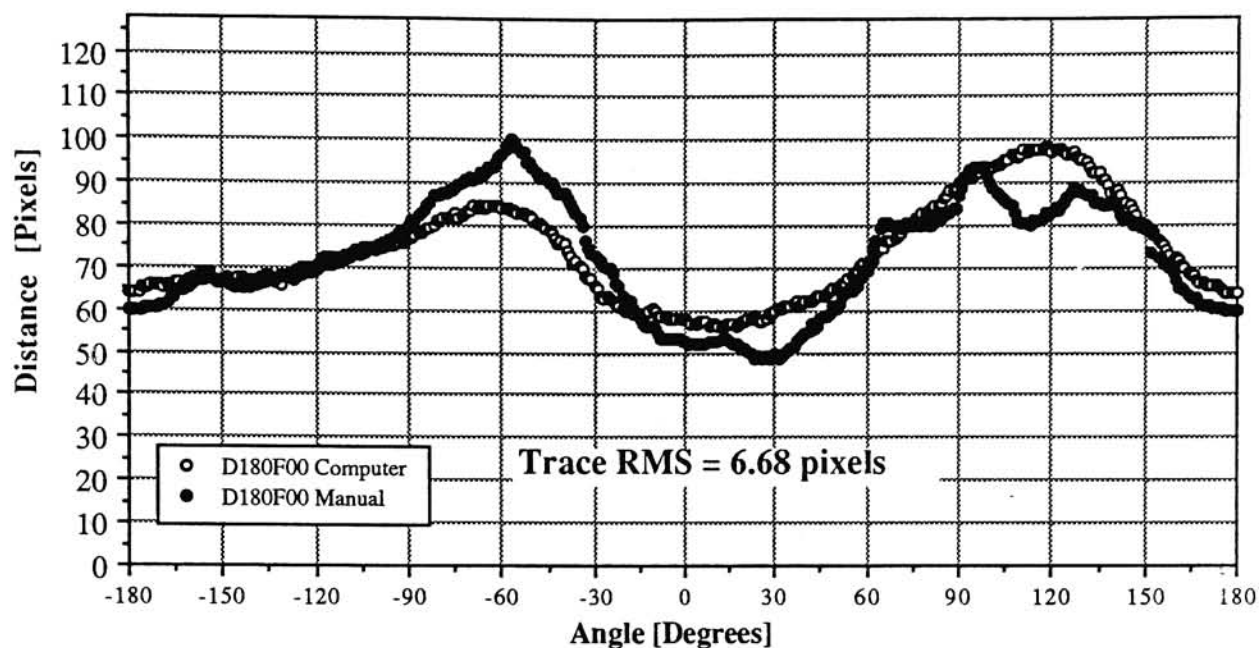


Figure 35a - D180F00 trace plots

Contour Distance from Centroid (Manual)- D180F03

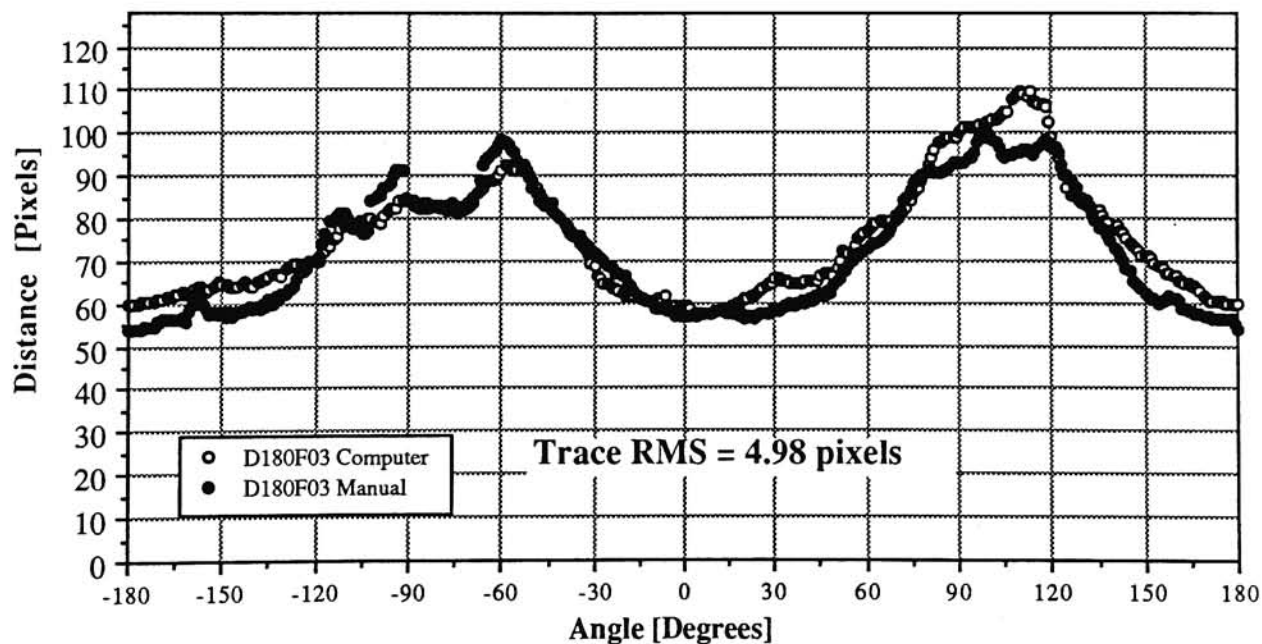


Figure 35b - D180F03 trace plots

Contour Distance from Centroid (Manual)- D180F09

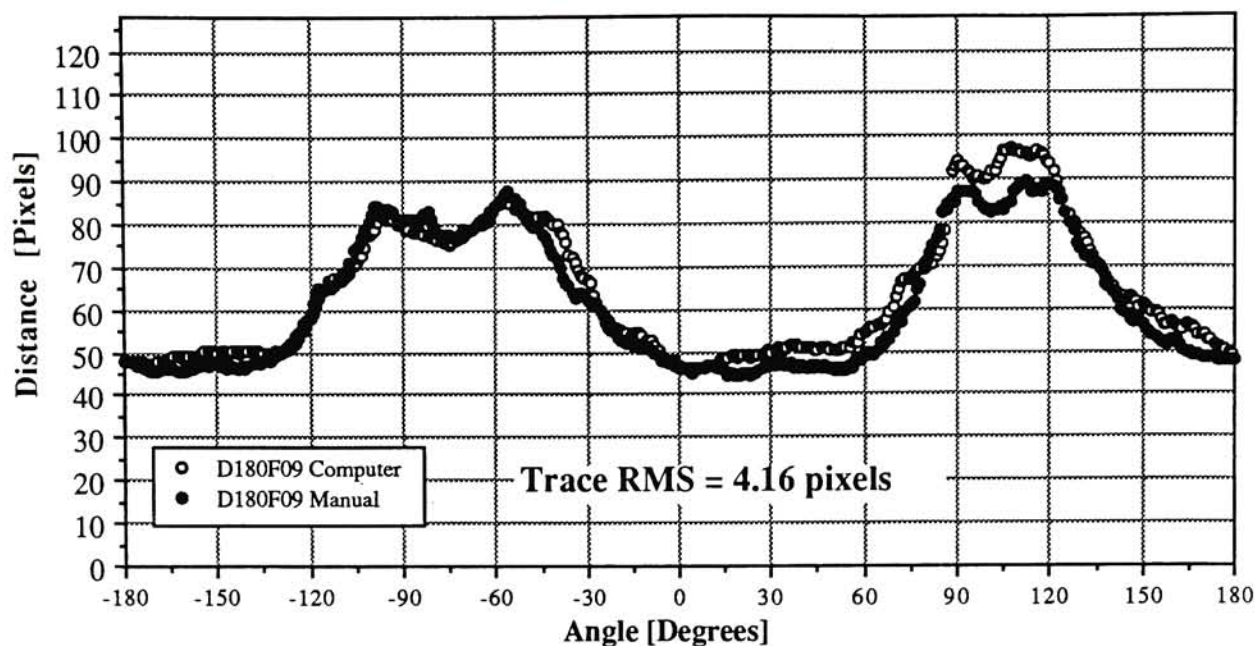


Figure 35c - D180F09 trace plots

Contour Distance from Centroid (Manual)- D180F12

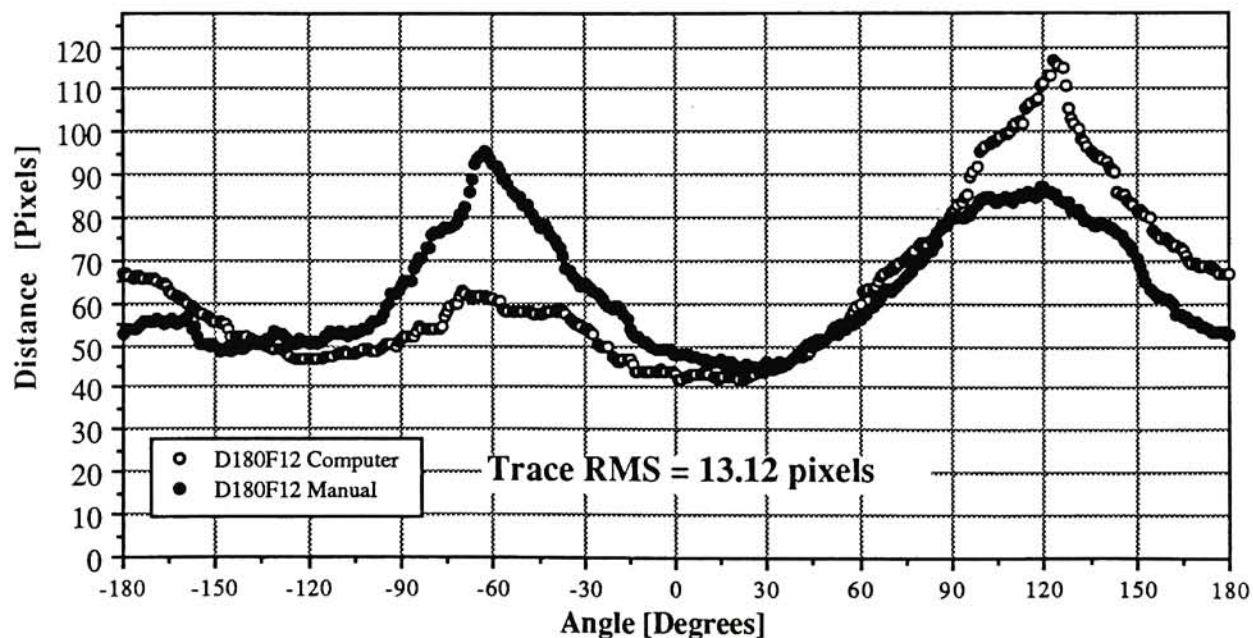


Figure 35d - D180F12 trace plots

Contour Distance from Centroid (Manual)- D180F18

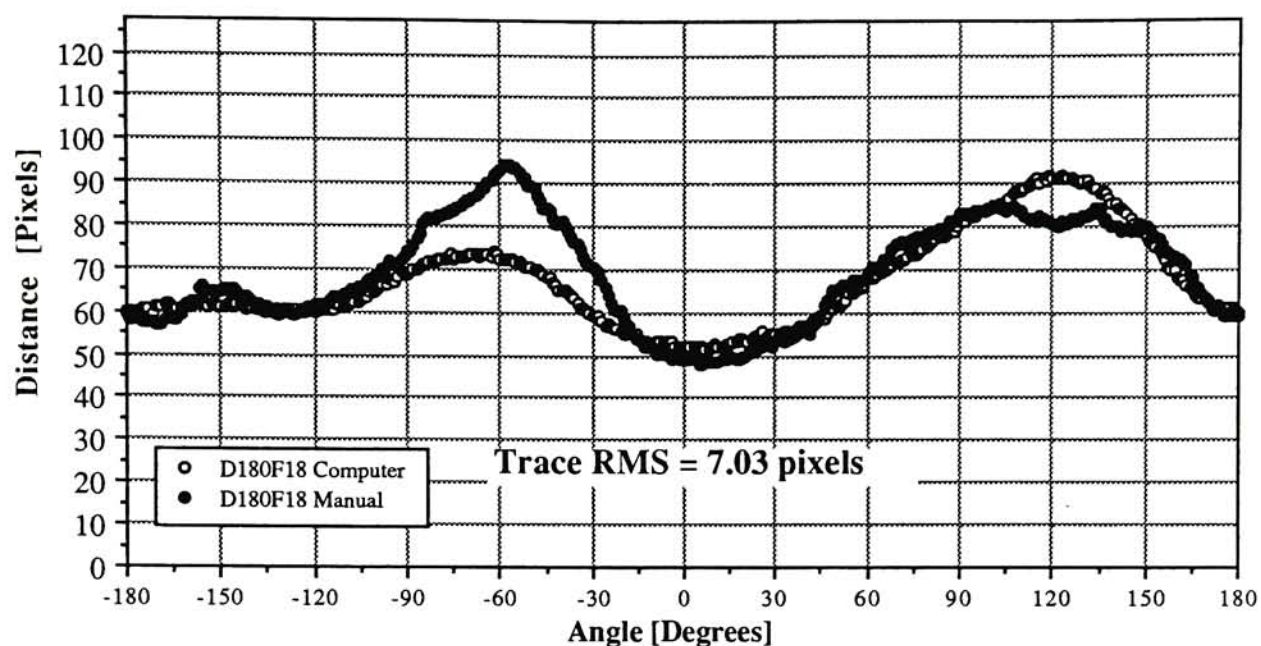


Figure 35e - D180F18 trace plots

Contour Distance from Centroid (Manual) - D180F22

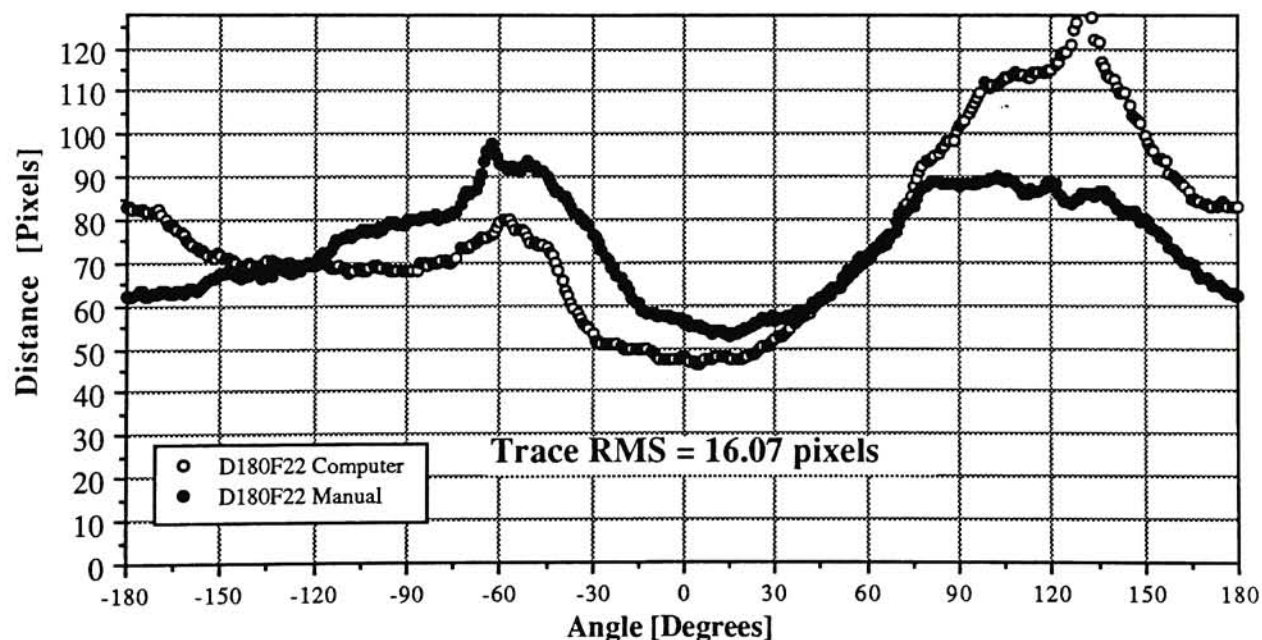


Figure 35f - D180F22 trace plots

The resulting RMS values and the corresponding manual/computer trace areas are summarized in **Table 3** along with the corresponding plot of computer contour area vs. manual contour area (**Figure 36**) and contour areas vs. image frame number (**Figure 37**).

Table 3 - Trace Statistics

Image	Trace RMS	Manual Trace Area	Computer Trace Area	%Error [Pixels^2]
D180F00	6.68	16997.0	17489.5	2.90
D180F03	4.98	17168.5	18328.5	6.76
D180F09	4.16	12732.5	13791.5	8.32
D180F12	13.12	13081.0	13487.0	3.10
D180F18	7.03	15515.0	14556.5	-6.18
D180F22	16.07	17450.5	19786.0	13.38

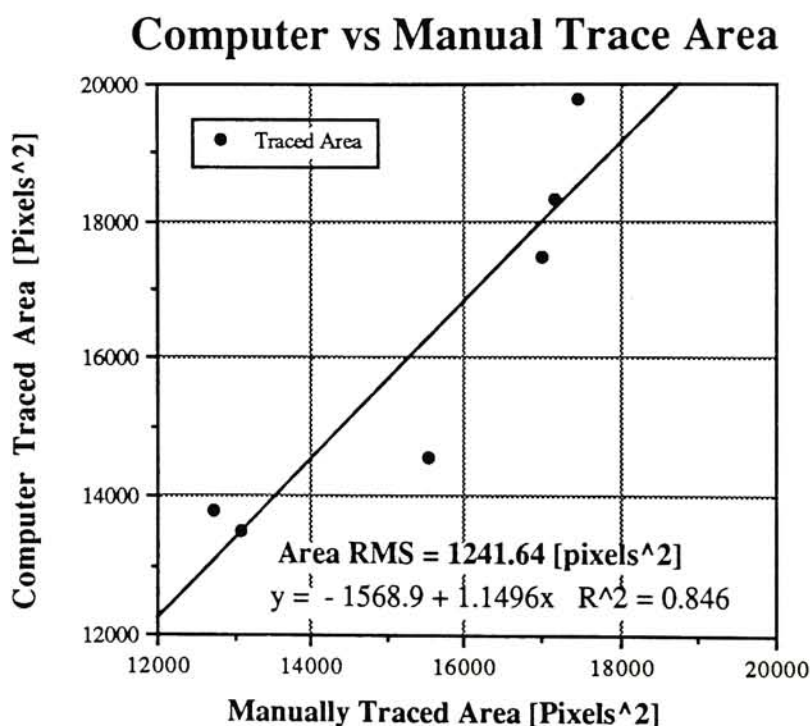


Figure 36

Ventricular Area vs Image Frame Number

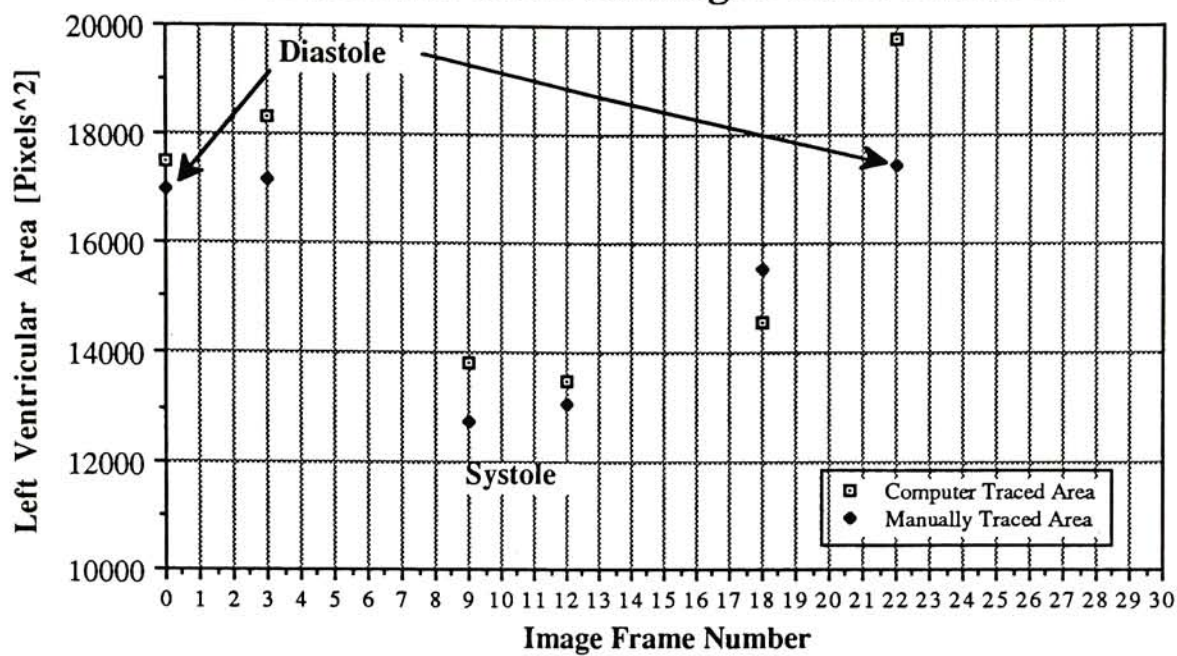


Figure 37

6. Conclusion:

Although the test image set is very small, the overall performance is encouraging as indicated by several of the final traces (D180F00, D180F03, D180F09, D180F18). A visual inspection of the final traces with the manual traces indicate a fair match. This is supported by the one-dimensional graphical representation of the traces as well as the resulting RMS calculations. The plot of the computer generated areas vs the manually generated areas shows that the computer generated traces have a tendency to overestimate the size of the ventricular chamber, as determined from the manual trace. The pattern and extent of this overestimation cannot be determined with any confidence due to the small test sample size. However, Schott *et al.*, 1986 have shown in a study of digital image processing algorithms for boundary enhancement that the areas of manual traces resulting from the processed image exhibited a tendency to underestimate the true cross sectional area of the left ventricle. This might indicate some possibilities that the computer generated trace may in fact perform better than a manually generated trace. On average, the ventricular area of the computer generated traces as a function of time followed the same diastole to diastole pattern as the manually generated trace as shown by **Figure 37**.

The two contour traces which exhibited anomalous results (D180F12 and D180F22) also corresponded to the two images which underwent several dilation/erosion iterations. Although the computer generated contour area bounded by these two traces have reasonably similar values compared to the manual trace areas, an overall shift of the computer generated trace away from the manually generated trace can be observed. This can be explained by the repeated dilation and erosion of the images using relatively small diameter circles as structuring elements. The quantized representations of circular structuring elements, especially at very small diameters, induces asymmetries due to aliasing. This asymmetry manifests itself as an anisotropic thickening and thinning effect resulting in the shifting of the contour traces and their centroids.

7. Recommendations:

The testing undertaken for this sequence of algorithms is far from conclusive. The trends exhibited by the results point to several fundamental issues warranting further investigation. The most critical of these is the optimum selection of structuring element shape, size, and dilation/erosion sequence iterations. Although the described morphological scheme provides a solution, it is by no means optimal. A similar optimum selection process can be made for the kernel size in the spatial convolution process.

As far as the implementation of the algorithms is concerned, an improvement that may be of merit involves a possible restructuring of the Robust Automatic Threshold Selector to operate in quad-tree/pyramid resolution mode in which the final image threshold is determined from thresholds computed in a piecewise manner from the subdivided areas of the image. This would probably require that the bit plane representation of the quad-tree multiresolution image be augmented to 8-bit image planes for each resolution level. This, at present, might not be feasible because it would require the image processing system to be configured as one workstation accessing all possible memory channels. Improvements might also be in order for the morphological operation routines since they were implemented directly from formal mathematical definitions. As with other implementations based on mathematical definitions, the resulting expressions and code may be elegant and compact, but the execution times are inherently slow and resource utilization inefficient. Possible hardware solutions should be investigated in the creation of the quad-tree/multiresolution structure since the algorithm is well suited to multi-processor configurations. Such a solution would make the data structure attractive to machine vision applications.

One interesting and potentially useful feature/signature that emerged from the analysis relates to the centroid distance vs. angle plot. To illustrate its utility, a simple shape such as an outline of a sector has been scaled into three different sizes (**Figure 38**), and a centroid distance vs angle plot was made for each of the sector outline (**Figure 39**). It is interesting to note how the shapes of the plots are similar to one another and how the critical points representing the corners of the outline all correspond to the same angles. By taking the maximum distances of each of the graphs and normalizing their values, all the graphs practically become the same plot as shown in **Figure 40**. This would indicate that traces of the same shape should generate the same normalized signature. **Figure 41** illustrating the effect of orientation by comparing the graphs of two of the sector outlines oriented 180° to each other. It shows that the signature simply becomes phase-shifted with respect to each other.

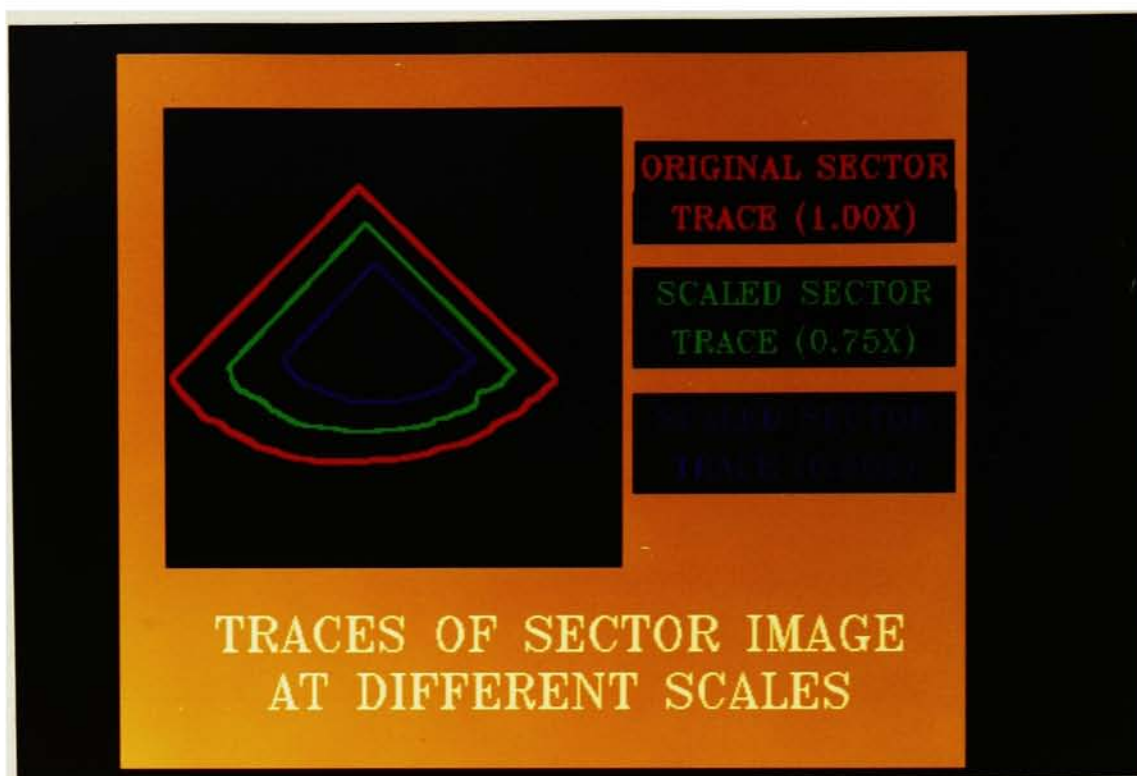


Figure 38 - Scaled sector outlines

Contour Distance from Centroid - Sector Outline

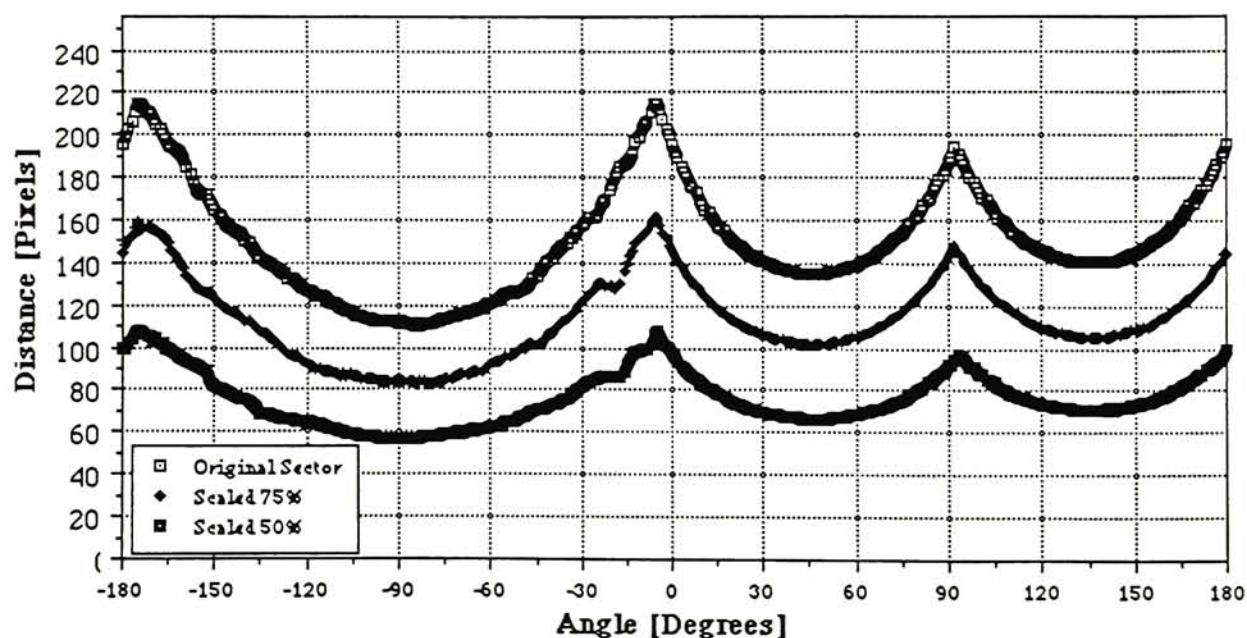


Figure 39 - Trace plots of scaled sector traces.

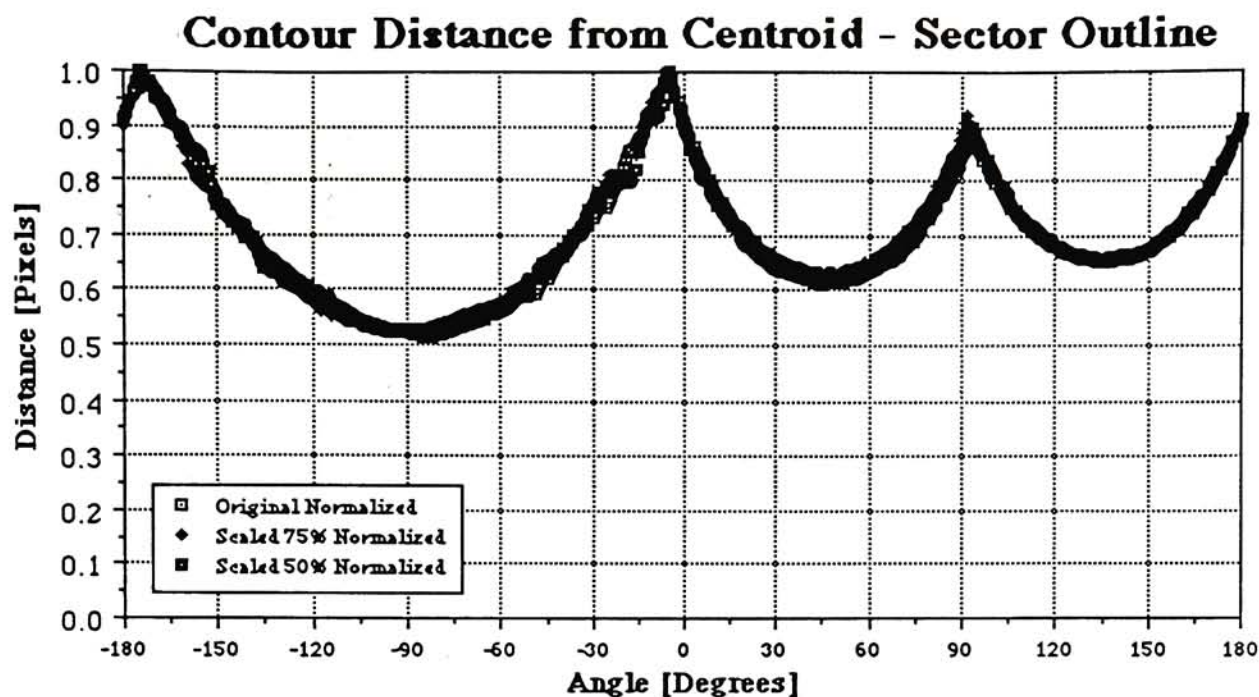


Figure 40 - Trace plots of normalized sector traces.

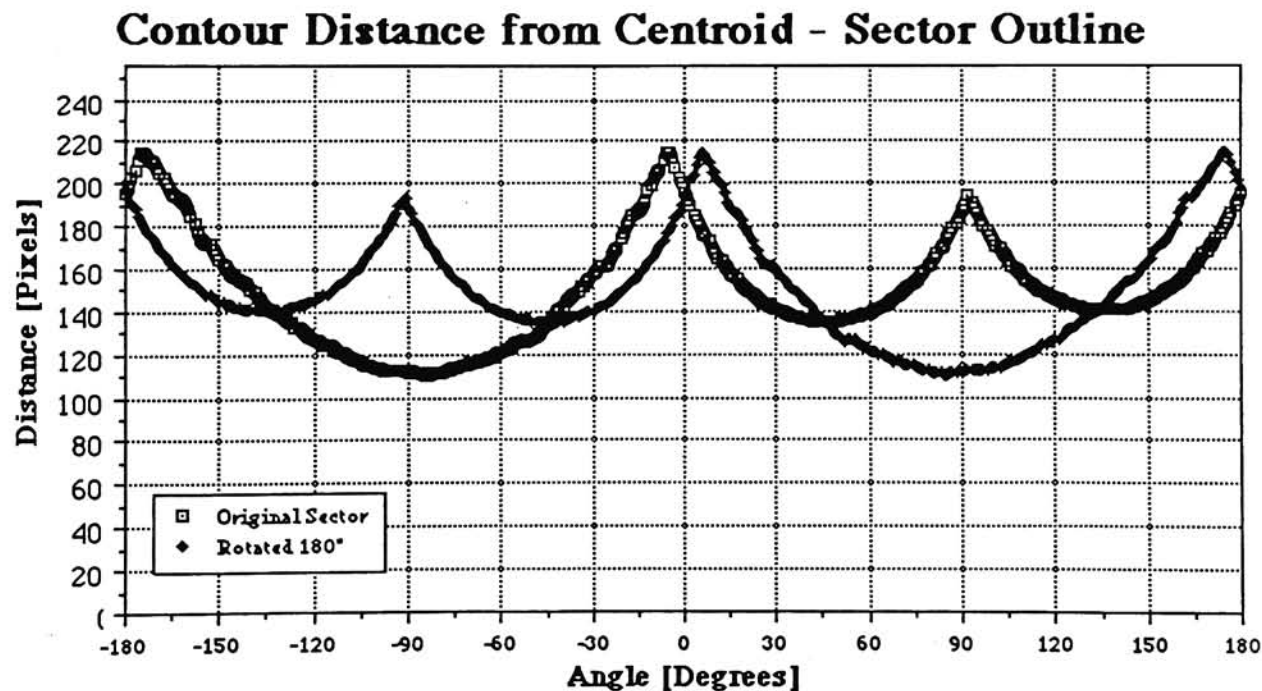


Figure 41 - Trace plots of rotated sector traces.

These results are very useful when applied to shape recognition because it generates a signature that is invariant of scaling which affect the performance of several traditional pattern recognition techniques. Since this signature is one-dimensional, its subsequent analysis using digital signal processing techniques such as auto-correlations and filterings should be greatly simplified. This feature along with the other algorithms implemented in this thesis provides several basic but necessary tools which can be adapted to other problems of image analysis.

References:

Adam D., *et al.*, "Semiautomated Border Tracking of Cine Echocardiographic Ventricular Images". **IEEE Transactions on Medical Imaging**. Volume MI-6, Number 3, September 1987.

Ballard D.H., Brown C.M., **Computer Vision**, Prentice-Hall, Edgewood Cliffs, NJ, 1982.

Brown, Owen R. , ed., **The Echo Digest**, September 1980, Volume 2, No. 1.

Chandra, K.B., *et al.*, "Three-dimensional Echocardiographic Reconstruction of the Intact Heart: Calculation of the Normal Diastolic Elastic Properties of the Canine Left Ventricle". Abstract submitted to The American Heart Associations 56th Scientific Sessions, November 14-17 1983.

Cooley, J.W., and J.W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," **Math. Comp.**, Volume 19, Number 90, pp. 297-301, 1965.

Cromwell L., Weibell F.J., Pfeiffer E.A., **Biomedical Instrumentation and Measurements**, Prentice Hall, Edgewood Cliffs, NJ, 1980.

Feigenbaum H., **Echocardiography** (3rd edit) Lea & Febiger, Philadelphia, PA, 1981.

Gaskill J.D., **Linear Systems, Fourier Transforms, and Optics**, John Wiley & Sons, New York, NY, 1978.

Giardina C.R., Dougherty E.R., **Morphological Methods in Image Processing**, Edgewood Cliffs, NJ, 1988.

Gonzalez R.C., Wintz P., **Digital Image Processing**, Addison-Wesley, Reading, MA, 1977.

Kittler J., Illingworth J., Fölgein J., "Threshold Selection Based on a Simple Image Statistic", **Computer Vision, Graphics, and Image Processing**. Volume 30, pp. 125-147, 1985.

Matsumoto M., *et al.*, "Three-dimensional Echocardiography for Spatial Visualization and Volume Calculation of Cardiac Structures", **J Clin Ultrasound**, 9:157-165, April 1981.

McCann H.A., *et al.*, "A Method for Three-Dimensional Ultrasonic Imaging of the Heart In Vivo". **Dynamic Cardiovascular Imaging**, Volume 1, Number 2, pp. 97-109, Summer 1987.

Nixon, J.V., *et.al.*, "Three-dimensional echoventriculography". **Clinical Investigations**, September 1983, Volume 106, Number 3.

Pavlidis T., **Algorithms for Graphics and Image Processing**, Computer Science Press, Rockville, MD, 1982.

Pratt W.K., **Digital Image Processing**, Wiley, New York, NY, 1978.

Sawada Hitoshi, *et al.* "Three dimensional reconstruction of the left ventricle from multiple cross sectional echocardiograms Value for measuring left ventricular volume". **Br Heart J** 1983; 50: 438-42

Schott J.R., Volchok W.J., Moos S., Sunni S., Ghosh A., Nanda N., "Comparison of Digital Image Processing Algorithms for Enhancement of Boundaries in Two-Dimensional Echocardiography", (abstr), **Clin Res**, 34:308A, 1986.

Schott J.R., *et al.*, "Feasibility Analysis of 3-D Reconstruction of Color Doppler Flow Velocity", **Journal of the American College of Cardiology**, 9(1987):165A.

Skorton D.J., *et al.*, "Digital Image Processing of Two-Dimensional Echocardiograms: Identification of the Endocardium". **The American Journal of Cardiology**, September 1981, Volume 48.

Zhu H., "Scatterer Number Density Estimation for Tissue Characterization in Ultrasound Imaging". M.S. Thesis, Center for Imaging Science, The Rochester Institute of Technology, August 1990.

Appendix A:

Result Table Summary

RESULT SUMMARY TABLE

<u>Image Name</u>	D180F00	D180F03	D180F09	D180F12	D180F18	D180F22
<u>Manual/Computer</u>	84	73	84	85	83	95
<u>Thresholds</u>	104	107	109	102	104	106
<u>[0:255]</u>						
<u>Error</u>	20	34	25	17	21	11
<u>%</u>	(23.81%)	(46.58%)	(29.76%)	(20.00%)	(25.30%)	(11.58%)
<u>Manual/Computer</u>	16997.00	17168.50	12732.50	13081.00	15515.00	17450.50
<u>Trace Area</u>	17489.50	18328.50	13791.50	13487.00	14556.50	19786.00
<u>[Pixels]</u>						
<u>Error</u>	492.50	1160.00	1059.00	406.00	-958.50	2335.50
<u>%</u>	(2.90%)	(6.76%)	(8.32%)	(3.10%)	-(6.18%)	(13.38%)
<u>Manual/Computer</u>	(219,294)	(220,290)	(220,290)	(210,292)	(213,299)	(217,293)
<u>Centroid</u>	(215,301)	(220,298)	(220,295)	(195,315)	(208,308)	(200,314)
<u>[Pixels]</u>						
<u>Error</u>	(+Δ4.0,-Δ7.0)	(+Δ0.0,-Δ8.0)	(+Δ0.0,-Δ5.0)	(+Δ15.0,-Δ23.0)	(+Δ5.0,-Δ9.0)	(+Δ17.0,-Δ21.0)
<u>Distance</u>	8.06	8.00	5.00	27.46	10.30	27.02
<u>Trace</u>	6.68	4.98	4.16	13.12	7.03	16.07
<u>RMS</u>						
<u>[Pixels]</u>						

Appendix B:
Main Program Listings of Algorithms

```

/* Start of multiresolution program *****/
*
*   This program creates a quad-tree representaion of a binary
*   image at a given starting resolution and location by calling
*   create_quad_tree.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       create_quad_tree
*       initialize_ip_no_clear
*
*   Modification History:
*
*   12/20/90  Completed documentation & comments - Rolando Raqueño
*
*****/

/*-----
FUNCTION PROTOTYPE SECTION-----
*/

void create_quad_tree(  short int x_position,
                       short int y_position,
                       short int window_size );

void initialize_ip_no_clear( void );

/*-----
CREATE_QUAD_TREE FUNCTION MAIN BODY-----
*/

main()
{
short int x_position = 0;      /* x-coordinate of lower left corner */
short int y_position = 0;      /* y-coordinate of lower left corner */
short int window_size = 512;   /* Starting size of area to process */

initialize_ip_no_clear();      /* Reset without clearing channels */

create_quad_tree(             x_position,
                             y_position,
                             window_size/2 );
}
/* End of create_quad_tree function *****/

```

PROGRAM Trace_Contour

```

*****
*
*   This program takes the thresholded image in multiresolution
*   form and tries to determine which resolution the cavity of the
*   echocardiogram becomes closed. Once this has been determined,
*   a morphological closing process consisting of a dilation
*   followed by an erosion will be applied using structuring
*   elements based on the size information determined from the
*   multiresolution structure. This process is repeated until a
*   closed contour is created. The processed image is then saved.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Initialize_IP
*       Show_Picture_8bit
*       Move_Channel_to_8bit_Buffer
*       Open_Contour
*       Clear_Channel
*       Draw_Circle
*       Reset_View_Channel
*       Clear_Overlay_Graphics_Channel
*       Dilate
*       Copy_Channel
*       Erode
*       Save_Picture_8bit
*
*   Modification History:
*
*   12/20/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80 input_file_name      ! File name of quad image.
CHARACTER*80 output_file_name     ! File name of result image.
INTEGER*2 Operand_Channel         ! Original Image.
INTEGER*2 Structuring_Element_Channel ! Structuring Element.
INTEGER*2 Result_Channel          ! Result of operations.
INTEGER*2 Structuring_Element_Grey_Level ! Structuring element
INTEGER*2 Structuring_Element_x_center ! characteristic
INTEGER*2 Structuring_Element_y_center ! specifications.
INTEGER*2 Structuring_Element_Width ! SE width argument.
INTEGER*2 Boundary_Modification_Factor ! Boudary error term.
INTEGER*2 Starting_x_lower_left_corner ! Starting coordinates of
INTEGER*2 Starting_y_lower_left_corner ! SE search space.
INTEGER*2 Starting_quadrant_search_size ! Starting search size.
INTEGER*2 First_x_position         ! Starting search point
INTEGER*2 First_y_position         ! for contour following.
INTEGER*2 x_neutral_axis           ! x location of Centroid.
INTEGER*2 y_neutral_axis           ! y location of Centroid.
INTEGER*2 Trace_Channel            ! Channel for Tracing.
INTEGER*2 Corrected_Trace_Channel ! Channel for final trace.
INTEGER*2 MAX_QUADRANT_LEVEL       ! Maximum depth of quad-tree
REAL*4 structuring_width_factor

```



```

PARAMETER ( Operand_Channel = 0 )
PARAMETER ( Structuring_Element_Channel = 1 )
PARAMETER ( Trace_Channel = 1 )
PARAMETER ( Result_Channel = 2 )
PARAMETER ( Corrected_Trace_Channel = 2 )
PARAMETER ( Structuring_Element_Grey_Level = 255 )
PARAMETER ( Structuring_Element_x_center = 255 )
PARAMETER ( Structuring_Element_y_center = 255 )
PARAMETER ( Starting_x_lower_left_corner = 0 )
PARAMETER ( Starting_y_lower_left_corner = 0 )
PARAMETER ( Starting_Quadrant_Search_Size = 256 )
PARAMETER ( First_x_position = 200 )
PARAMETER ( First_y_position = 300 )
PARAMETER ( MAX_QUADRANT_LEVEL = 6 )

INTEGER*2 Quadrant_Level           ! Level in quad-tree.
INTEGER*2 Quadrant_Pixel_Size      ! Size of quadrant pixels.
INTEGER*2 Dilation_Width           ! Width of SE for dilation.
INTEGER*2 Erosion_Width            ! Width of SE for erosion.
REAL*4   Area_of_Contour( 0: MAX_QUADRANT_LEVEL )
BYTE     Quad_Tree_Buffer( 0:511, 0:511 ) ! 8-bit Image buffer.
REAL*4   Open_Contour

DATA Area_of_Contour/ -1,MAX_QUADRANT_LEVEL*-1 /

structuring_width_factor = SQRT(2.0)
Boundary_Modification_Factor = 0

TYPE *, 'Input the file name of image to process >'
ACCEPT ('(A80)'), input_file_name

TYPE *, 'Input the file name of result image >'
ACCEPT ('(A80)'), output_file_name

TYPE *, 'Structuring width factor = ', structuring_width_factor
TYPE *, ' '

CALL Initialize_IP
CALL Show_Picture_8bit( Input_file_name, Operand_Channel )
CALL Move_Channel_to_8bit_Buffer( Operand_Channel,
                                   Quad_Tree_Buffer )

Quadrant_Level = 0 ! Image at full resolution.
Quadrant_Pixel_Size=2**Quadrant_Level ! Size of quadrant pixel.

Area_of_Contour( Quadrant_Level ) =
.      Open_Contour( Trace_Channel,
.                    Quad_Tree_Buffer,
.                    Quadrant_Pixel_Size,
.                    First_x_position,
.                    First_y_position,
.                    x_neutral_axis,
.                    y_neutral_axis )

TYPE *, ' Quad Level = ', Quadrant_Level, ' Area = ',
.      Area_of_Contour( Quadrant_Level )
TYPE *, ' Pixel Size = ', Quadrant_Pixel_Size
TYPE *, ' Centroid = (', x_neutral_axis, ', ', y_neutral_axis, ' ) '
TYPE *, ' '

```

```

CALL Clear_Channel( Trace_Channel )

DO WHILE (      Area_of_Contour( 0 ) .LT. 0.0
              .AND.
              Area_of_Contour( 1 ) .LT. 0.0      )

    DO WHILE (      Quadrant_Level .LE. MAX_QUADRANT_LEVEL
                  .AND.
                  Area_of_Contour(Quadrant_Level) .LT. 0.0 )

        Quadrant_Level = Quadrant_Level + 1
        Quadrant_Pixel_Size = 2 ** Quadrant_Level

        Area_of_Contour( Quadrant_Level ) =
            Open_Contour( Trace_Channel,
                          Quad_tree_buffer,
                          Quadrant_Pixel_Size,
                          First_x_position,
                          First_y_position,
                          x_neutral_axis,
                          y_neutral_axis )

        TYPE *, ' Quad Level = ', Quadrant_Level, ' Area = ',
                Area_of_Contour( Quadrant_Level )
        TYPE *, ' Pixel Size = ', Quadrant_Pixel_Size
        TYPE *, ' Centroid = (', x_neutral_axis, ', ',
                y_neutral_axis, ' ) '
        TYPE *, ' '

    CALL Clear_Channel ( Trace_Channel )

END DO

CALL Clear_Channel( Structuring_Element_Channel )

Structuring_Element_Width = NINT( Quadrant_Pixel_Size *
                                   structuring_width_factor )

CALL Draw_Circle (      Structuring_Element_Channel,
                        Structuring_Element_Grey_Level,
                        Structuring_Element_x_center,
                        Structuring_Element_y_center,
                        Structuring_Element_Width)

IF ( Quadrant_Level .LT. 2 ) THEN

    CALL Reset_View_Channel
    CALL Clear_Overlay_Graphics_Channel

    TYPE *, 'Dilating Image by a circle with diameter = ',
            Structuring_Element_Width
    TYPE *, ' '

    Boundary_Modification_Factor =
        Boundary_Modification_Factor +
        Structuring_Element_Width

    CALL Dilate( %val( Structuring_Element_Grey_Level ),
                %val( Starting_x_lower_left_corner ),
                %val( Starting_y_lower_left_corner ),
                %val( Starting_Quadrant_Search_Size ) )

ELSE

```

```

CALL Reset_View_Channel
CALL Clear_Overlay_Graphics_Channel

TYPE *, 'Dilating Image by a circle with diameter = ',
      Structuring_Element_Width
TYPE *, ' '

Boundary_Modification_Factor =
      Boundary_Modification_Factor +
      Structuring_Element_Width

CALL Dilate( %val( Structuring_Element_Grey_Level ),
            %val( Starting_x_lower_left_corner ),
            %val( Starting_y_lower_left_corner ),
            %val( Starting_Quadrant_Search_Size ) )

CALL Reset_View_Channel
CALL Clear_Overlay_Graphics_Channel

CALL Copy_Channel( Result_Channel, Operand_Channel )
CALL Clear_Channel( Structuring_Element_Channel )

Structuring_Element_Width =
      NINT( Quadrant_Pixel_Size/2 *
            structuring_width_factor )

CALL Draw_Circle (      Structuring_Element_Channel,
                        Structuring_Element_Grey_Level,
                        Structuring_Element_x_center,
                        Structuring_Element_y_center,
                        Structuring_Element_Width)

TYPE *, 'Eroding Image by a circle with diameter = ',
      Structuring_Element_Width
TYPE *, ' '

Boundary_Modification_Factor =
      Boundary_Modification_Factor -
      Structuring_Element_Width

CALL Erode( %val( Structuring_Element_Grey_Level ),
            %val( Starting_x_lower_left_corner ),
            %val( Starting_y_lower_left_corner ),
            %val( Starting_Quadrant_Search_Size ) )

END IF

CALL Reset_View_Channel
CALL Clear_Overlay_Graphics_Channel

CALL Copy_Channel( Result_Channel, Operand_Channel )
CALL Clear_Channel( Result_Channel )

CALL Move_Channel_to_8bit_Buffer(      Operand_channel,
                                     Quad_Tree_Buffer )

TYPE *, ' '
TYPE *, ' '
TYPE *, ' Check processed image '
TYPE *, ' '

Quadrant_Level = 0
Quadrant_Pixel_Size = 2 ** Quadrant_Level

```

```

Area_of_Contour( Quadrant_Level ) =
    Open_Contour( Trace_Channel,
                  Quad_tree_buffer,
                  Quadrant_Pixel_Size,
                  First_x_position,
                  First_y_position,
                  x_neutral_axis,
                  y_neutral_axis )

TYPE *, ' Quad Level = ', Quadrant_Level, ' Area = ',
    Area_of_contour( Quadrant_Level )
TYPE *, ' Pixel Size = ', Quadrant_Pixel_Size
TYPE *, ' Centroid = (', x_neutral_axis, ', ', y_neutral_axis, ' ) '
TYPE *, ' Boundary Modification Factor = ',
    Boundary_Modification_Factor
TYPE *, ' '

CALL Clear_Channel( Trace_Channel )

Quadrant_Level = 1
Quadrant_Pixel_Size = 2 ** Quadrant_Level
Area_of_Contour( Quadrant_Level ) =
    Open_Contour( Trace_Channel,
                  Quad_tree_buffer,
                  Quadrant_Pixel_Size,
                  First_x_position,
                  First_y_position,
                  x_neutral_axis,
                  y_neutral_axis )

TYPE *, ' Quad Level = ', Quadrant_Level, ' Area = ',
    Area_of_contour( Quadrant_Level )
TYPE *, ' Pixel Size = ', Quadrant_Pixel_Size
TYPE *, ' Centroid = (', x_neutral_axis, ', ', y_neutral_axis, ' ) '
TYPE *, ' Boundary Modification Factor = ',
    Boundary_Modification_Factor
TYPE *, ' '

CALL Clear_Channel( Trace_Channel )

END DO

CALL Clear_Channel ( Trace_Channel )

CALL Save_Picture_8bit( Output_File_Name, Operand_Channel )

END ! Trace_Contour

```


program final_contour

```

*****
*
*   This program takes the morphologically processed closed image
*   and determines the final corrected contour of the boundary
*   trace. The resulting trace points are written to a user
*   specified file. As of this test version, it is the
*   responsibility of the user to keep track and enter the
*   correction factor for the final trace. This correction factor
*   is discussed in greater detail in final_corrected_contour.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Initialize_IP
*       Show_Picture_8bit
*       Move_Channel_to_8bit_Buffer
*       Open_Contour
*       Clear_Channel
*       Final_Corrected_Contour
*
*   Modification History:
*
*   12/20/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

implicit none

```

character*80 final_image_file_name      ! Image file name.
character*80 output_trace_file_name     ! Output trace file.
integer*2 final_image_channel           ! Channel to store image.
integer*2 window_size                   ! Size variable.
integer*2 resolution_level              ! Resolution in Quad-Tree.
integer*2 first_x_position              ! Starting position of the
integer*2 first_y_position              ! contour following kernel
integer*2 x_neutral_axis                ! Resulting centroid
integer*2 y_neutral_axis                ! coordinates.
integer*2 correction_factor             ! Trace correction factor
integer*2 vector_channel                ! Channel to display
parameter ( vector_channel = 1 )         ! uncorrected trace.
integer*2 corrected_vector_channel      ! Channel to display
parameter ( corrected_vector_channel = 2 ) ! corrected trace.

real*4 contour_area                     ! Area of contour trace.
real*4 open_contour                     ! Open contour function.
real*4 final_corrected_contour          ! Final corrected contour.
parameter ( final_image_channel = 0 )    ! Display image to 0.
parameter ( first_x_position = 200 )    ! Contour following kernel
parameter ( first_y_position = 300 )    ! starting position.
byte buffer(0:511,0:511)               ! Buffer for image.

type *, 'Input file name of final image >'
accept ('(a80)'), final_image_file_name
type *, 'Input file name of output trace file >'
accept ('(a80)'), output_trace_file_name
type *, 'Input the final image resolution level >'

```

```

accept *, resolution_level
type *, 'Input the correction factor >'
accept *, correction_factor

call initialize_ip
call show_picture_8bit( final_image_file_name, final_image_channel )
call move_channel_to_8bit_buffer( final_image_channel, buffer )

window_size = 2 ** resolution_level

contour_area = open_contour(      vector_channel,
.                                buffer,
.                                window_size,
.                                first_x_position,
.                                first_y_position,
.                                x_neutral_axis,
.                                y_neutral_axis )

type *, ' window_size = ', window_size
type *, ' contour_area = ', contour_area
type *, ' x_neutral_axis = ', x_neutral_axis
type *, ' y_neutral_axis = ', y_neutral_axis
type *, ' '

call clear_channel (vector_channel )
call clear_channel (corrected_vector_channel)

contour_area = Final_corrected_contour( output_trace_file_name,
.                                       vector_channel,
.                                       corrected_vector_channel,
.                                       buffer,
.                                       window_size,
.                                       x_neutral_axis,
.                                       y_neutral_axis,
.                                       correction_factor )

type *, ' window_size = ', window_size
type *, ' contour_area = ', contour_area
type *, ' x_neutral_axis = ', x_neutral_axis
type *, ' y_neutral_axis = ', y_neutral_axis
type *, ' '

end ! Final_contour.

```

Appendix C:

Subprogram Listings of Algorithms

SUBROUTINE

Add_16bit

! 16-bit Addition.

```
*****
*
*   This subroutine calculates a 16-bit pixel-by-pixel addition
*   using the DVP. The first addend image will be in channel (0,1).
*   The second addend image will be in channel (2,3). The resulting
*   16-bit sum image will be deposited in channel (2,3) overwriting
*   the second addend image.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Attach_DVP
*       Initialize_DVP
*       DVP_Add_16bit
*       Detach_DVP
*
*   Modification History:
*
*   7/23/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

CALL Attach_DVP
CALL Initialize_DVP
CALL DVP_Add_16bit
CALL Detach_DVP

! Get the DVP for use.
! Initialize the DVP.
! Start 16-bit DVP Operation.
! Release DVP resource.

RETURN

END ! Add_16bit


```

SUBROUTINE      And_8bit(      Operand1_Channel,
.                  Operand2_Channel,
.                  Result_Channel ) ! Logical Image AND

```

```

*****
*
*      This subroutine performs an 8-bit image AND of two images
*      in two channels and placing the result in a third channel.
*      The subroutine utilizes the DVP to execute the bitwise
*      logical AND.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*          Attach_DVP
*          Initialize_DVP
*          DVP_AND_8bit
*          Detach_DVP
*
*      Modification History:
*
*      4/6/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INTEGER*2 Operand1_Channel      ! Channel Number of first image.
INTEGER*2 Operand2_Channel      ! Channel Number of second image.
INTEGER*2 Result_Channel        ! Channel Number of result image.

```

```

CALL Attach_DVP                  ! Allocate the DVP.
CALL Initialize_DVP              ! Initialize and Reset the DVP.

```

```

CALL DVP_And_8bit(      Operand1_Channel,
.                  Operand2_Channel,
.                  Result_Channel)      ! 8-bit DVP image AND.

```

```

CALL Detach_DVP                  ! Deallocate the DVP.

```

RETURN

END ! AND_8bit.

```
REAL*4 FUNCTION Area_under_Vector(      Array_of_Points )
```

```
*****
*
*   This function returns the full resolution x-coordinate of the
*   center point of the quadrant pixel ( i.e. super-pixel ) having
*   window_size by window_size dimensions [full resolution pixels]
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*
*   Modification History:
*
*   12/13/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

```
IMPLICIT NONE
```

```
INTEGER*2      Array_of_Points( 0:1, 0:1 )
INTEGER*2      x0                                ! x vector tail.
INTEGER*2      y0                                ! y vector tail.
INTEGER*2      x1                                ! x vector head.
INTEGER*2      y1                                ! y vector head.
REAL*4         Delta_x                           ! Δx.
REAL*4         Delta_y                           ! Δy.
REAL*4         Area_under_Rectangle
REAL*4         Area_under_Triangle

x0 =  Array_of_Points(0,0)
y0 =  Array_of_Points(1,0)
x1 =  Array_of_Points(0,1)
y1 =  Array_of_Points(1,1)

Delta_x =      x1 - x0                            ! Δx.
Delta_y =      y1 - y0                            ! Δy.

Area_under_Rectangle =  y0 * Delta_x              ! Base*Height.
Area_under_Triangle =  0.5 * Delta_x * Delta_y    ! 0.5*Base*Height.
Area_under_Vector = Area_under_Rectangle + Area_under_Triangle

RETURN

END ! Area_Under_Vector
```

SUBROUTINE Assign_DVP_Output_Channel(DVP_Bus, Output_Channel)

```
*****
*
* This subroutine attaches a specific DVP output bus to a
* given channel.
*
* Image Processing Interface Library (IPI) call
* developed by S. Schultz
*
* Written by Rolando Raqueño (Center for Imaging Science)
*
* Include Files:
* IPI_IPILIB
*
* Called Routines:
* IPI_DVPOutput
*
* Modification History:
*
* 7/20/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB'

```
INTEGER*4 Status      ! Status call variable.
INTEGER*4 Unit        ! IP8500 unit number.
INTEGER*2 DVP_bus     ! DVP Bus to attach.
INTEGER*2 Output_Channel ! Output channel to attach.
```

Common /IPI_Unit/ Unit ! IPI_Unit common block.

```
Status = Ipi_DVPOutput( Unit, DVP_Bus, Output_Channel )
CALL Ipi_ErrorCheck( Status, 'Error Assigning DVP Output' )
```

RETURN

END ! Assign_DVP_Output_Channel

SUBROUTINE Attach_IP ! Attach the IP8500.

```
*****
*
*   This subroutine attaches the IP8500 using an IPI library call.
*
*   Image Processing Interface Library (IPI) call
*       developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_AttUnit
*       IPI_ErrorCheck
*
*   Modification History:
*
*   4/5/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.

INTEGER*4 Unit ! IP8500 unit number.

Common /IPI_Unit/ Unit ! IPI_Unit common block.

Status = Ipi_AttUnit(Unit) ! Attach Unit IPI call.

IF (.NOT.Ipi_ErrorCheck(Status,'Error Attaching Deanza Unit')) Stop

RETURN

END ! Attach_Ip

SUBROUTINE Attach_DUP ! Allocate the DUP for exclusive use

```
*****
*
*   This subroutine attaches the DUP by an IPI library call.
*
*   Image Processing Interface Library (IPI) call
*       developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_AttDUP
*       IPI_ErrorCheck
*
*   Modification History:
*
*   4/7/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INCLUDE 'IPI_IPILIB'                ! Include IPI header file.

INTEGER*4 Status                    ! Status call variable.
INTEGER*4 Unit                      ! IP8500 unit number.
INTEGER*2 Wait                      ! Wait flag in case of busy DUP.

PARAMETER (Wait = 1)                ! Wait for a busy DUP.

COMMON /IPI_Unit/ Unit              ! IPI_Unit common block.

Status = Ipi_AttDUP( Unit, Wait ) ! Attach DUP call.

CALL Ipi_ErrorCheck( Status, 'Error Attaching DUP' )

RETURN

END ! Attach_DUP.
```

SUBROUTINE Calculate_Histogram(Channel) ! Compute Histogram

```
*****
*
*   This subroutine signals the fast histogram board to start
*   calculating the histogram of the image in the specified
*   channel. The fast histogram buffer should be cleared with
*   the Clear_Histogram call before calculating the histogram.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_CalcHst
*       IPI_ErrorCheck
*
*   Modification History:
*
*   8/3/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

```
IMPLICIT NONE
INCLUDE 'IPI_IPILIB'            ! Include IPI header file.

INTEGER*4 Status                ! Status call variable.
INTEGER*4 Unit                 ! IP8500 unit number.
INTEGER*4 Channel               ! Channel to histogram.

COMMON /Ipi_Unit/ Unit           ! IPI_Unit common block.

Status = Ipi_CalcHst( Unit, Channel )   ! Calculate histogram.
CALL Ipi_ErrorCheck ( Status, 'Error Calculating Histogram' )

RETURN

END ! Calculate_Histogram
```

SUBROUTINE Calculate_Histogram_ROI(Channel) ! Calculate Histogram

```

*****
*
*   This subroutine calculates the histogram for a predefined
*   region of interest in a given channel. The region of
*   interest is defined by a preceding Region_of_Interest call.
*   The resulting histogram array containing the probability
*   distributions resides in the histogram buffer of the
*   fast histogram board. The histogram array can be read from
*   hardware by the Get_Histogram subroutine.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_CalcHst
*       IPI_ErrorCheck
*
*   Modification History:
*
*   5/30/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INCLUDE 'IPI_IPILIB'           ! Include IPI header file.

INTEGER*4 Status                ! Status call variable.
INTEGER*4 Unit                  ! IP8500 unit number.
INTEGER*4 Channel               ! Channel to be histogrammed.

COMMON /Ipi_Unit/ Unit          ! Ipi_Unit common block.

Status = Ipi_CalcHst( Unit, Channel, Ipi_M_UseRegion )

CALL Ipi_ErrorCheck ( Status, 'Error Calculating Histogram' )

RETURN

END ! Calculate_Histogram_ROI

```

LOGICAL*1 FUNCTION Channel_empty(Channel)

```
*****
*
*   The following function checks to see if a specified channel is
*   empty, i.e., all pixels in the channel are of zero value. The
*   routine calls quadrant_pixel_code which checks the absence or
*   presence of pixels in a given region of interest.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       quadrant_pixel_code
*
*   Modification History:
*
*   12/13/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*2 Channel           ! Channel to check for empty.
INTEGER*2 x_llc             ! x-position of lower left corner.
INTEGER*2 y_llc             ! y-position of lower left corner.
INTEGER*2 Window_size       ! Size of window to check.
```

```
PARAMETER( x_llc = 0 )
PARAMETER( y_llc = 0 )
PARAMETER( Window_size = 512 ) ! Check 512x512 window.
```

```
INTEGER*2 quadrant_pixel_status_code ! Result of check.
INTEGER*2 quadrant_pixel_code        ! integer*2 function.
```

```
quadrant_pixel_status_code = quadrant_pixel_code(channel,
.                               x_llc,
.                               y_llc,
.                               window_size)
```

```
Channel_empty = (quadrant_pixel_status_code .eq. 1 )
```

RETURN

END ! Channel_Empty

SUBROUTINE Clear_Channel_Extended(Channel_Number) ! Clear a channel

```
*****
*
*   This subroutine clears individual 1024 x 1024 image channels
*   of the IP8500 by an IPI Library call.
*
*   Image Processing Interface Library (IPI) call
*       developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_ClearChan
*       IPI_ErrorCheck
*
*   Modification History:
*
*   4/5/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI Header File.

INTEGER*4 Status ! Status call variable.

INTEGER*4 Unit ! IP8500 Unit Number.

INTEGER*2 Channel_Number ! Image Channel Number to Clear.

COMMON /Ipi_Unit/ Unit ! Ipi_Unit common block.

Status = IPI_ClearChan(Unit, Channel_Number, IPI_M_ExtMem) ! Call.

CALL Ipi_ErrorCheck(Status, 'Error Clearing Channel')

RETURN

END ! Clear_Channel_Extended

```

SUBROUTINE CLEAR_CHANNEL( CHANNEL_NUMBER )      ! Clear sector 0

*****
*
*   This subroutine clears sector 0 of a 512 x 512 image channel.
*
*   Image Processing Interface Library (IPI) call
*       developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_ClearChan
*       IPI_ErrorCheck
*
*   Modification History:
*
*   7/23/90  Completed documentation & comments - Rolando Raqueño
*
*****

IMPLICIT NONE

INCLUDE 'IPI_IPILIB'      ! Include IPI Header File.
INTEGER*4 Status          ! Status call variable.
INTEGER*4 Unit            ! IP8500 Unit Number.
INTEGER*2 Channel_Number  ! Image channel number for clear.

COMMON /Ipi_Unit/ Unit    ! IPI_Unit common block.

STATUS = IPI_CLEARCHAN( UNIT, CHANNEL_NUMBER )
CALL IPI_ERRORCHECK( STATUS, 'Error Clearing Channel' )

RETURN

END ! Clear_Channel

```

SUBROUTINE Clear_Channels ! Clear sector 0 for all.

```
*****
*
* This subroutine clears out all the channels from 0 to 3
* for sector zero of each channel of the IP8500 system.
*
* Written by Rolando Raqueño (Center for Imaging Science)
*
* Include Files:
*
* Called Routines:
*      Clear_Channel
*
* Modification History:
*
* 7/23/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INTEGER*2 Channel_Number

DO Channel_Number = 0,3

CALL Clear_Channel(Channel_Number)

END DO

RETURN

END ! Clear_Channels

SUBROUTINE Clear_Channels_Extended ! Clear channels/sectors.

```
*****
*
*   This subroutine clears out all the channels from 0 to 3 and
*   all the sectors of each channel of the IP8500 system.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channel_Extended
*
*   Modification History:
*
*   4/5/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INTEGER*2 Channel_Number ! Channel Number variable.

DO Channel_Number = 0,3 ! Clear channels/sector 0 to 3.

call Clear_Channel_Extended(Channel_Number) ! Clear

END DO

Return

END ! Clear_Channels_Extended

SUBROUTINE Clear_Histogram

!Clear the histogram buffer

```
*****
*
*   The following subroutine clears the histogram buffer residing
*   in the fast histogram board. This call should be made
*   before each histogram calculation since the buffers are not
*   cleared automatically.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_ClearHst
*       IPI_ErrorCheck
*
*   Modification History:
*
*   5/30/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.

INTEGER*4 Unit ! IP8500 unit number.

COMMON /IPI_Unit/Unit ! IPI_Unit common block.

Status = Ipi_ClearHst(Unit) ! Clear histogram board buffer.

CALL Ipi_ErrorCheck(Status, 'Error clearing histogram')

RETURN

END ! Clear_histogram

SUBROUTINE Clear_Scroll ! Reset to No Scroll.

```
*****
*
*      This subroutine initializes the scroll registers of the
*      IP8500 system by defining the X and Y pixel location of
*      the upper left corner pixel of the image on the screen
*      via a Scroll_Channel call.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*          Scroll_Channel
*
*      Modification History:
*
*      4/5/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel_Mask            ! Bitwise channel selection variable.
INTEGER*2 X_Value                ! X Scroll Register Variable.
INTEGER*2 Y_Value                ! Y Scroll Register Variable.
```

```
PARAMETER (Channel_Mask = 15)    ! Select all 15 channels if present.
PARAMETER (X_Value = 0)          ! Define upper left x = 0.
PARAMETER (Y_Value = 511)        ! Define upper left y = 511.
```

```
CALL Scroll_Channel( X_Value, Y_Value, Channel_Mask ) ! Scroll Channel
```

RETURN

END ! Clear_Scroll

SUBROUTINE Clear_Overlay_Graphics_Channel

```
*****
*
*   This subroutine clears the overlay graphics channel.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channel
*
*   Modification History:
*
*   9/27/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INTEGER*2 Overlay_Graphics_Channel ! Gaphics Channel.

PARAMETER (Overlay_Graphics_Channel = 3) ! Channel 3.

CALL Clear_Channel(Overlay_Graphics_Channel) ! Clear channel.

RETURN

END ! Clear_Overlay_Graphics_Channel

SUBROUTINE Constant_A_DUP(Constant) ! Set DUP constant A.

```
*
*      This subroutine sets the DUP constant input A to a specified
*      constant. This becomes the input into the DUP in lieu of a
*      memory channel.
*
*      Image Processing Interface Library (IPI) call
*      developed by S. Schultz
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*          IPI_IPILIB
*
*      Called Routines:
*          IPI_DUPConstant
*          IPI_ErrorCheck
*
*      Modification History:
*
*      8/13/90  Completed documentation & comments - Rolando Raqueño
*
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.

INTEGER*4 Unit ! IP8500 unit number.

INTEGER*2 Constant ! Constant value for DUP input.

COMMON /Ipi_Unit/ Unit ! IPI_Unit common block.

Status = Ipi_DUPConstant(Unit, Constant) ! Set DUP Input A.
CALL Ipi_ErrorCheck(Status, 'Error Constant A DUP')

RETURN

END ! Constant_A_DUP


```

SUBROUTINE Constant_Channel_OR_ROI(      Grey_Level,
.                                         Channel,
.                                         X_LLC,
.                                         Y_LLC,
.                                         X_URC,
.                                         Y_URC      ) ! Constant OR ROI

```

```

*
*      This subroutine performs a bitwise logical OR operation between
*      a specified region of interest of a channel and a constant
*      area bounded by the same region of interest. This routine
*      allows right-hand coordinate. The Grey_Level value,
*      Channel number, and bounding coordinates are passed on to the
*      Constant_Channel_ROI routine which deposits a constant value to
*      a rectangular Region Of Interest.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*          Attach_DUP
*          Initialize_DUP
*          Region_of_Interest
*          Constant_A_DUP
*          DUP_OR_8bit
*          Detach_DUP
*
*      Modification History:
*
*      8/14/90  Completed documentation & comments - Rolando Raqueño
*
*
*****

```

IMPLICIT NONE

```

INTEGER*2 Grey_Level      ! Grey level value of ROI.
INTEGER*2 Channel         ! Channel number of ROI area
INTEGER*2 X_LLC           ! Lower-left hand x-coordinate of ROI.
INTEGER*2 Y_LLC           ! Lower-left hand y-coordinate of ROI.
INTEGER*2 X_URC           ! Upper-right hand x-coordinate of ROI.
INTEGER*2 Y_URC           ! Upper-right hand y-coordinate of ROI.

```

```

CALL Attach_DUP           ! Get the DUP for use.
CALL Initialize_DUP       ! Initialize the DUP.
CALL Region_of_Interest( X_LLC, Y_LLC, X_URC, Y_URC ) ! Set ROI.
CALL Constant_A_DUP( Grey_Level ) ! Set DUP Input Constant A.
CALL DUP_OR_8bit( Channel, Channel, Channel ) ! OR channel with A.
CALL Detach_DUP          ! Release DUP resource

```

RETURN

END ! Constant_Channel_OR_ROI

```

SUBROUTINE      Copy_Channel(  Source_Channel,
                               Destination_Channel)

```

```

*****
*
*      This subroutine copies sector 0 of a 512 x 512 image channel
*      specified by the source_channel argument to sector 0 of the
*      image channel specified by the destination_channel argument.
*
*      Image Processing Interface Library (IPI) call
*      developed by S. Schultz
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*          IPI_IPILIB
*
*      Called Routines:
*
*          Clear_Scroll
*          IPI_CopyChan
*          IPI_ErrorCheck
*
*      Modification History:
*
*      9/29/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INCLUDE 'IPI_IPILIB'           ! Include IPI Header File.
INTEGER*4 Status                ! Status call variable.
INTEGER*4 Unit                  ! IP8500 Unit Number.
INTEGER*2 Source_Channel        ! Source channel to copy.
INTEGER*2 Destination_Channel   ! Destination channel of copy.

```

```

COMMON /Ipi_Unit/ Unit          ! IPI_Unit common block.

```

```

CALL Clear_Scroll              ! Clear scroll before copying.

```

```

Status = Ipi_CopyChan( Unit,
                        Source_Channel,
                        Destination_Channel)

```

```

CALL Ipi_ErrorCheck( Status, 'Error Copying Channel' )

```

```

Return

```

```

END ! Copy_Channel

```

```

/* Start of create_quad_tree function *****/
*
*   This function creates a quad-tree representaion of a binary
*   image at a given starting resolution and location. Because the
*   original image is binary, it is possible to represent the
*   subsequent low resolution images as binary images also.
*
*   This means that it is possible to create, represent, and
*   store a quad-tree representation of a 512x512 image in 8 1-bit
*   image planes ignoring the two images representing the lowest
*   resolutions, i.e., the most significant image bit plane will
*   contain the 512x512 image representation, the next most
*   significant bit will contain the 256x256 image, the next will
*   contain the 128x128 image, etc. Since a 2x2 and a 1x1 image
*   contains little information, they can be ignored.
*
*   The process starts by rescursively subdividing the image and
*   taking a fast histogram of the quadrant region of interest.
*   This is accomplished by the quadrant_pixel_status routine.
*   The absence and presence of pixels determine the values of the
*   super-pixels represented by each quadrant. The values are
*   deposited into proper bit-plane by the quadrant_pixel_mask
*   routine.
*
*   The x_llc and y_llc indicate the position of the lower left
*   corner of the region to be processed into a quad-tree.
*   The argument window_size, indicates the dimension, in full
*   resolution pixels, comprising the starting super-pixel.
*
*   The four quadrants are relatively positioned in the following
*   (x,y) scheme.
*
*
*   (0,1)  (1,1)
*   (0,0)  (1,0)
*
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       quadrant_pixel_status
*       quadrant_pixel_mask
*
*   Modification History:
*
*   8/11/90  Completed documentation & comments - Rolando Raqueño
*   9/24/90  Updated quadrant_pixel_mask argument list
*           - Rolando Raqueño
*
*****/

```

[illegible]


```

quadrant_pixel_mask(    result_channel,
                        quadrant_pixel_status_code,
                        x_position,
                        y_position,
                        window_size ); /* Assign binary
                                        value to
                                        quadrant
                                        pixel.
                                        */

switch ( quadrant_pixel_status_code )
{
    case 1:             /* Quadrant pixel is all 0's */
        break;          /* go to next quadrant pixel */

    case 2:             /* Quadrant pixel is all 1's */
        break;          /* go to next quadrant pixel */

    case 3:             /* Quadrant pixel is 1's and
                        0's. Recurse to smaller
                        size quadrant pixel.      */

        create_quad_tree(    x_position,
                            y_position,
                            window_size/2);

        break;

    default:
        printf("Error in Creating Quad Tree");
}
}
}

/* End of create_quad_tree function *****/

```

SUBROUTINE Derivative_0_Degree_16bit(Input_File)

```

*****
*
*   This subroutine calculates a 16-bit derivative of the kernel
*
*           -1  0  1
*
*   using the DVP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channels
*       Load_First_Operand_16bit
*       Scroll_Operand1_East_16bit
*       Scroll_Operand1_West_16bit
*       Add_16bit
*       Subtract_16bit
*       Store_Result_16bit
*
*   Modification History:
*
*   7/27/90 Completed documentation & comments - Rolando Raqueño
*   8/2/90  Used Store_Result_16bit routine - Rolando Raqueño
*   8/7/90  Removed 2nd Load_First_Operand_16bit call from routine.
*           - Rolando Raqueño
*   8/7/90  Removed redundant call to Clear_Scroll routine
*           - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80  Input_File           ! 16-bit operand image.
CHARACTER*80  Output_File          ! Scratch file image.

Output_File = 'Derivative.0'       ! Name of scratch file.

CALL Clear_Channels                 ! Clear all channels.
CALL Load_First_Operand_16bit( Input_File ) ! Load image.
CALL Scroll_Operand1_East_16bit ! Scroll Right 1 pixel.
CALL Add_16bit                      ! Add scrolled image.
CALL Scroll_Operand1_West_16bit ! Scroll Left 1 pixel.
CALL Subtract_16bit                 ! Subtract images as 16 bits.
CALL Store_Result_16bit( Output_File ) ! Save intermediate result.

```

RETURN

END ! Derivative_0_Degree_16bit

SUBROUTINE Derivative_45_Degree_16bit(Input_File)

```

*****
*
*   This subroutine calculates a 16-bit derivative of the kernel
*
*
*           -1   0   1
*           0   0   0
*           1   0   1
*
*   using the DVP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channels
*       Load_First_Operand_16bit
*       Scroll_Operand1_SouthEast_16bit
*       Scroll_Operand1_NorthWest_16bit
*       Add_16bit
*       Subtract_16bit
*       Store_Result_16bit
*
*   Modification History:
*
*   7/27/90 Completed documentation & comments - Rolando Raqueño
*   8/2/90  Used Store_Result_16bit routine - Rolando Raqueño
*   8/7/90  Removed 2nd Load_First_Operand_16bit call from routine
*           - Rolando Raqueño
*   8/7/90  Removed redundant call to Clear_Scroll routine
*           - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80   Input_File           ! 16-bit operand image.
CHARACTER*80   Output_File          ! Scratch file image.

Output_File = 'Derivative.45'       ! Name of scratch file.

CALL Clear_Channels                  ! Clear all channels.
CALL Load_First_Operand_16bit( Input_File ) ! Load image.
CALL Scroll_Operand1_SouthEast_16bit ! Scroll Down-Right 1 pixel.
CALL Add_16bit                       ! Add scrolled image.
CALL Scroll_Operand1_NorthWest_16bit ! Scroll Up-Left 1 pixel.
CALL Subtract_16bit                  ! Subtract images as 16 bits.
CALL Store_Result_16bit( Output_File ) ! Save intermediate result.

```

RETURN

END ! Derivative_45_Degree_16bit

SUBROUTINE Derivative_90_Degree_16bit(Input_File)

```

*****
*
*   This subroutine calculates a 16-bit derivative of the kernel
*
*
*           -1
*           0
*           1
*
*   using the DVP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channels
*       Load_First_Operand_16bit
*       Scroll_Operand1_North_16bit
*       Scroll_Operand1_South_16bit
*       Add_16bit
*       Subtract_16bit
*       Store_Result_16bit
*
*   Modification History:
*
*   7/27/90 Completed documentation & comments - Rolando Raqueño
*   8/2/90   Used Store_Result_16bit routine - Rolando Raqueño
*   8/7/90   Removed 2nd Load_First_Operand_16bitcall from routine.
*           - Rolando Raqueño
*   8/7/90   Removed redundant call to Clear_Scroll routine
*           - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80   Input_File           ! 16-bit operand image.
CHARACTER*80   Output_File          ! Scratch file image.

```

```

Output_File = 'Derivative.90'       ! Name of scratch file.

```

```

CALL Clear_Channels                  ! Clear all channels.
CALL Load_First_Operand_16bit( Input_File ) ! Load image.
CALL Scroll_Operand1_South_16bit      ! Scroll Down 1 pixel.
CALL Add_16bit                        ! Add scrolled image.
CALL Scroll_Operand1_North_16bit      ! Scroll Up 1 pixel.
CALL Subtract_16bit                  ! Subtract images as 16 bits.
CALL Store_Result_16bit( Output_File ) ! Save intermediate result.

```

RETURN

END ! Derivative_90_Degree_16bit

SUBROUTINE Derivative_135_Degree_16bit(Input_File)

```

*****
*
*   This subroutine calculates a 16-bit derivative of the kernel
*
*
*           -1
*          0
*         1
*
*   using the DVP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channels
*       Load_First_Operand_16bit
*       Scroll_Operand1_Southwest_16bit
*       Scroll_Operand1_Northeast_16bit
*       Add_16bit
*       Subtract_16bit
*       Store_Result_16bit
*
*   Modification History:
*
*   7/27/90 Completed documentation & comments - Rolando Raqueño
*   8/2/90   Used Store_Result_16bit routine - Rolando Raqueño
*   8/7/90   Removed 2nd Load_First_Operand_16bit call from routine
*           - Rolando Raqueño
*   8/7/90   Removed redundant call to Clear_Scroll routine
*           - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80   Input_File           ! 16-bit operand image.
CHARACTER*80   Output_File          ! Scratch file image.

```

```

Output_File = 'Derivative.135'      ! Name of scratch file.

```

```

CALL Clear_Channels                  ! Clear all channels.
CALL Load_First_Operand_16bit( Input_File ) ! Load image.
CALL Scroll_Operand1_SouthWest_16bit ! Scroll Down-Left 1 pixel.
CALL Add_16bit                       ! Add scrolled image.
CALL Scroll_Operand1_NorthEast_16bit ! Scroll Up-Right 1 pixel.
CALL Subtract_16bit                  ! Subtract images as 16 bits
CALL Store_Result_16bit( Output_File ) ! Save intermediate result.

```

RETURN

END ! Derivative_135_Degree_16bit

SUBROUTINE Detach_DUP ! Detach DUP after use.

```
*****
*
* This subroutine detaches the DUP by an IPI library call.
*
* Image Processing Interface Library (IPI) call
* developed by S. Schultz
*
* Written by Rolando Raqueño (Center for Imaging Science)
*
* Include Files:
* IPI_IPILIB
*
* Called Routines:
* IPI_DetDUP
* IPI_ErrorCheck
*
* Modification History:
*
* 4/9/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.
INTEGER*4 Unit ! IP8500 unit number.

COMMON /Ipi_Unit/ Unit ! IPI_Unit common block.

Status = Ipi_DetDUP (Unit) ! Detach DUP call.

CALL Ipi_ErrorCheck(Status, 'Error Detaching DUP')

RETURN

END ! Detach_DUP

[illegible]


```

y_position,
window_size);

break;

case 3:
    dilate( bit_level,
            x_position,
            y_position,
            window_size / 2 );
    break;

default:
    printf("Error in Dilation");
}
}
}
}
}
/* End of dilate function *****/

```

```
subroutine dilate_pixel( x, y ) ! Dilate the image by a single pixel.
```

```
*****
*
* This subroutine dilates an image in channel 0 by a single
* pixel of the structuring element found in channel 1. The
* resulting intermediary contribution of this erosion is applied
* to the result channel (channel 2).
*
* Written by Rolando Raqueño (Center for Imaging Science)
*
* Include Files:
*
*
* Called Routines:
*           Scroll_Channel
*           OR_8bit
*
* Modification History:
*
* 9/27/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

```
implicit none
```

```
integer*2 x           ! structuring element pixel x-position.
integer*2 y           ! structuring element pixel y-position.
integer*4 channel_mask ! Bitwise channel selection variable.
integer*2 x_scroll    ! x-scroll register variable.
integer*2 y_scroll    ! y-scroll register variable.
integer*2 operand1_channel ! Channel number of first image operand.
integer*2 result_channel ! Channel number of result image.
```

```
parameter( operand1_channel = 0 ) ! First image in channel 0.
parameter( result_channel = 2 ) ! Result image in channel 2.
parameter( channel_mask = 2 ** operand1_channel ) ! Scroll channel 2.
```

```
x_scroll = -x + 256 ! Origin of unscrolled image is (256,256)
                  ! with x_scroll and y_scroll representing
                  ! the image coordinates of upper left corner
y_scroll = -( 512 - y ) + 767 ! after scroll.
```

```
call scroll_channel( x_scroll, y_scroll, channel_mask ) ! Scroll call.
```

```
call OR_8bit( operand1_channel,
              result_channel,
              result_channel ) ! OR Image.
```

```
return
```

```
end ! Dilate_Pixel
```

```

/* Start of dilate_quadrant_pixel function *****/
*
*   This function is used as a means to establish auto variables
*   which are referenced by the FORTRAN subroutine dilate_square.
*   This is necessary in order to allow recursive functions to
*   call this routine and maintain unique parameter values stored
*   in the stack during recursive invocation of this function.
*   Without this function, recursive calls to erode_square cannot
*   be made because of FORTRAN's inherent limitation of passing all
*   parameters by reference, and its lack of dynamic storage
*   allocation.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       dilate_square
*
*   Modification History:
*
*   9/27/90  Completed documentation & comments - Rolando Raqueño
*
*****/

/*-----
FUNCTION PROTOTYPE SECTION-----*/
extern void dilate_square(    unsigned short int *ptr_x_position,
                             unsigned short int *ptr_y_position,
                             unsigned short int *ptr_window_size );

/*-----
DILATE_QUADRANT_PIXEL MAIN BODY-----*/
void dilate_quadrant_pixel(    unsigned short int x_position,
                             unsigned short int y_position,
                             unsigned short int window_size )
{
    dilate_square(    &x_position,
                     &y_position,
                     &window_size    );
}
/* End of dilate_quadrant_pixel function *****/

```

```

subroutine dilate_square(      x_lower_left_corner,
                             y_lower_left_corner,
                             window_size )

```

```

*****
*
*   This subroutine dilates an image in channel 0 by a square
*   section of the structuring element found in channel 1. The
*   resulting intermediary contribution of this dilation is applied
*   to the result channel (channel 2).
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Dilate_Pixel
*
*   Modification History:
*
*   9/27/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

implicit none

```

```

integer*2 x_lower_left_corner      ! x of lower left corner.
integer*2 y_lower_left_corner      ! y of lower left corner.
integer*2 window_size              ! size of square.

```

```

integer*2 x                        ! x coordinate of pixel.
integer*2 y                        ! y coordinate of pixel.

```

```

integer*2 x_upper_right_corner     ! x of upper right corner.
integer*2 y_upper_right_corner     ! y of upper right corner.

```

```

x_upper_right_corner = x_lower_left_corner + window_size - 1
y_upper_right_corner = y_lower_left_corner + window_size - 1

```

```

do x = x_lower_left_corner, x_upper_right_corner
  do y = y_lower_left_corner, y_upper_right_corner

```

```

    call dilate_pixel( x, y )      ! Dilate by a pixel

```

```

  end do

```

```

end do

```

```

return

```

```

end ! Dilate_Square

```


SUBROUTINE Disable_ITT(Channel) ! Bypass ITT's

```
*****
*
*   The following subroutined disables the ITT's by calling
*   the Enable_ITT routine and sending disabling flags in
*   the Memory_or_VOC and Memory_or_DVP variables
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Enable_ITT
*
*   Modification History:
*
*   4/6/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel                    ! Channel number to Disable.
INTEGER*2 VOC_ITT_Control           ! Control ITT between memory and VOC
INTEGER*2 DVP_ITT_Control           ! Control ITT between memory and DVP
```

```
PARAMETER (VOC_ITT_Control = 0) ! Disable VOC ITT.
PARAMETER (DVP_ITT_Control = 0) ! Disable DVP ITT.
```

```
CALL Enable_ITT(Channel, VOC_ITT_Control, DVP_ITT_Control)
```

RETURN

END ! Disable_ITT

SUBROUTINE Disable_ITTs ! Bypass ITT's.

```
*****
*
*        The following subroutined disables the ITT's by calling
*        the Disable_ITT routine and passing it the appropriate
*        channel numbers from 0 to 3. Both VOC and DUP ITT's
*        are disabled.
*
*        Written by Rolando Raqueño (Center for Imaging Science)
*
*        Include Files:
*
*        Called Routines:
*                        Disable_ITT
*
*        Modification History:
*
*        4/6/90   Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INTEGER*4 Channel ! Channel to disable VOC/DUP ITT

DO Channel = 0,3 ! Loop through the channels.

 CALL Disable_ITT(Channel) ! Disable ITT call.

END DO

RETURN

END ! Disable_ITTs

```

SUBROUTINE Draw_Circle(
Channel,
Intensity,
x_center,
y_center,
Diameter )

*****
*      This subroutine draws a circle of a given diameter and center      *
*      with a specified grey level in a given channel.                      *
*                                                                              *
*      Written by Rolando Raqueño (Center for Imaging Science)             *
*                                                                              *
*      Include Files:                                                        *
*                                                                              *
*      Called Routines:                                                      *
*          Draw_Vector                                                        *
*                                                                              *
*      Modification History:                                                 *
*                                                                              *
*      9/14/90  Completed documentation & comments - Rolando Raqueño      *
*****

```

IMPLICIT NONE

```

INTEGER*2 Intensity
INTEGER*2 Channel
INTEGER*2 x_center
INTEGER*2 y_center
INTEGER*2 Diameter
INTEGER*2 Radius
INTEGER*2 x
INTEGER*2 x_first
INTEGER*2 x_last
INTEGER*2 delta_y
INTEGER*2 x0
INTEGER*2 y0
INTEGER*2 x1
INTEGER*2 y1
INTEGER*2 Array_of_Points( 0:1, 0:1)
EQUIVALENCE( Array_of_Points(0,0), x0 )
EQUIVALENCE( Array_of_Points(1,0), y0 )
EQUIVALENCE( Array_of_Points(0,1), x1 )
EQUIVALENCE( Array_of_Points(1,1), y1 )

Radius = Diameter / 2                ! Calculate radius.
x_first = x_center - Radius          ! Minimum x-point.
x_last = x_center + Radius           ! Maximum x-point.

DO x = x_first, x_last
    delta_y= NINT(SQRT(REAL(RADIUS)**2.0-REAL(x-x_center)**2.0))
    x0 = x
    y0 = y_center + delta_y
    x1 = x
    y1 = y_center - delta_y
    CALL DRAW_VECTOR(Channel, Intensity, Array_of_Points )
END DO

RETURN

END ! Draw_Circle

```

```

SUBROUTINE      Draw_Vector(      Channel,
.                                     Grey_Level,
.                                     Array_of_Points )

```

```

*****
*
*   This subroutine draws a vector of a specified grey level in a
*   given channel.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Draw_Vectors
*
*   Modification History:
*
*   9/14/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INTEGER*2      Number_of_Vectors      ! Number of Vectors.
PARAMETER      (Number_of_Vectors = 1)
INTEGER*2      Grey_Level              ! Vector grey level.
INTEGER*2      Array_of_Points(2, Number_of_Vectors + 1 ) ! Points.
INTEGER*2      Channel                 ! Channel for Vector.

```

```

CALL      Draw_Vectors(      Channel,
.                               Grey_Level,
.                               Number_of_Vectors,
.                               Array_of_Points      )

```

```

RETURN

```

```

END ! Draw_Vector

```



```

SUBROUTINE      Draw_Vectors(  Channel,
                               Grey_Level,
                               Number_of_Vectors,
                               Array_of_Points )

```

```

*****
*      This subroutine allows a set of vectors of a given grey level      *
*      to be drawn in the channels selected using the Channel_Mask        *
*      register. The number of vectors to be drawn is specified by        *
*      the Number_of_Vectors argument and the points defining the        *
*      vectors are contained in the array Array_of_Points. Drawing        *
*      multiple vectors assumes that the Array_of_Points contains a      *
*      starting point (tail of the first vector) followed by the          *
*      ending point (head of the first vector) of the first vector.      *
*      Subsequent vectors are drawn starting from the head of the        *
*      previously drawn vectors. This continues until the specified      *
*      number of vectors have been drawn.                                  *
*                                                                           *
*      Image Processing Interface Library (IPI) call                      *
*      developed by S. Schultz                                             *
*                                                                           *
*      Written by Rolando Raqueño (Center for Imaging Science)          *
*                                                                           *
*      Include Files:                                                     *
*                                                                           *
*          IPI_IPILIB                                                     *
*                                                                           *
*      Called Routines:                                                  *
*                                                                           *
*          IPI_ImageVector                                                *
*          IPI_ErrorCheck                                                 *
*                                                                           *
*      Modification History:                                              *
*                                                                           *
*      9/12/90  Completed documentation & comments - Rolando Raqueño    *
*****

```

```

IMPLICIT NONE
INCLUDE 'IPI_IPILIB'
INTEGER*4 Status
INTEGER*4 Unit
INTEGER*2 Channel
INTEGER*2 Grey_Level
INTEGER*4 Channel_Mask
INTEGER*2 Number_of_Vectors
INTEGER*2 Array_of_Points( 2, Number_of_Vectors + 1 )

! Include IPI header file.
! Status call variable.
! IP8500 unit number.
! Channel to Draw Vectors.
! Vector Grey Value.
! Bitwise channel selection.
! Number of Vectors.

COMMON /Ipi_Unit/ Unit
! IPI_Unit common block.

Channel_Mask = 2**Channel

Status = Ipi_ImageVector( Unit,
                          Channel_Mask,
                          Grey_Level,
                          Number_of_Vectors,
                          Array_of_Points )

CALL Ipi_ErrorCheck( Status, 'Error Drawing Vectors' )

RETURN

END ! Draw_Vectors

```

```

SUBROUTINE      DUP_And_8bit(  Operand1_Channel,
                              Operand2_Channel,
                              Result_Channel )

*****
*
*   This subroutine calls an IPI routine to execute an 8-bit DUP
*   image logical bitwise AND operation.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*       IPI_IPILIB
*
*   Called Routines:
*
*       IPI_DUPMath
*       IPI_ErrorCheck
*
*   Modification History:
*
*       12/11/90  Completed documentation & comments - Rolando Raqueño
*
*****
IMPLICIT NONE

INCLUDE 'IPI_IPILIB'                ! Include IPI header file.

INTEGER*4 Status                    ! Status call variable.
INTEGER*4 Unit                      ! IP8500 unit number.

INTEGER*2 Operand1_Channel          ! First operand channel.
INTEGER*2 Operand2_Channel          ! Second operand channel.
INTEGER*2 Result_Channel            ! Result channel.

COMMON /Ipi_Unit/ Unit              ! IPI_Unit common block.

Status = Ipi_DUPMath ( Unit,
                      Ipi_And,
                      Operand1_Channel,
                      Operand2_Channel,
                      Result_Channel )

CALL Ipi_ErrorCheck( Status, 'Error ANDing Channels' )

RETURN

END ! DUP_AND_8bit

```

SUBROUTINE DVP_Add_16bit ! 16-bit signed addition.

```

*****
*
*   This subroutine calls an IPI routine to execute a 16-bit DVP
*   image addition operation.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*       IPI_IPILIB
*
*   Called Routines:
*
*       IPI_DVPAdvMath
*       IPI_ErrorCheck
*
*   Modification History:
*
*   7/23/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

```

INTEGER*4 Status ! Status call variable.
INTEGER*4 Unit ! IP8500 unit number.
INTEGER*2 First_Addend_HI_Channel ! HI byte of addend (BB).
INTEGER*2 First_Addend_LO_Channel ! LO byte of addend (BA).
INTEGER*2 Second_Addend_HI_Channel ! HI byte of addend (9D).
INTEGER*2 Second_Addend_LO_Channel ! LO byte of addend (BC).
INTEGER*2 Sum_HI_Channel ! HI byte of sum (DVPB).
INTEGER*2 Sum_LO_Channel ! LO byte of sum (DVPA).
PARAMETER (First_Addend_HI_Channel = 0)
PARAMETER (First_Addend_LO_Channel = 1)
PARAMETER (Second_Addend_HI_Channel = 2)
PARAMETER (Second_Addend_LO_Channel = 3)
PARAMETER (Sum_HI_Channel = 2)
PARAMETER (Sum_LO_Channel = 3)

```

COMMON /IPI_Unit/ Unit ! IPI_Unit Common block.

```

Status = Ipi_DVPAdvMath( Unit, Ipi_Plus,
    First_Addend_HI_Channel,
    First_Addend_LO_Channel,
    Second_Addend_HI_Channel,
    Second_Addend_LO_Channel,
    Sum_HI_Channel,
    Sum_LO_Channel )

```

CALL Ipi_ErrorCheck(Status, 'Error Adding Channels')

RETURN

END ! DVP_Add_16bit

SUBROUTINE DVP_Maximum_16bit ! 16-bit signed DVP maximum.

```
*****
*
*   This subroutine calls an IPI routine to execute a 16-bit DVP
*   image signed pixel-by-pixel maximum operation.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_DVPMinMax
*
*   Modification History:
*
*   7/20/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.

INTEGER*4 Unit ! IP8500 unit number.

INTEGER*2 Mode ! Mode of maximum operation.

COMMON /IPI_Unit/ Unit ! IPI_Unit Common block.

PARAMETER (Mode = 3) ! 16-bit signed maximum.

Status = Ipi_DVPMinMax (Unit, Ipi_Maximum, Mode)

CALL Ipi_ErrorCheck(Status, 'Error on 16-bit Maximum Operation')

RETURN

END ! DVP_Maximum_16bit

SUBROUTINE DVP_Multiply_8bit ! 8-bit signed multiplication

```
*****
*
*   This subroutine calls an IPI routine to execute a DVP 8-bit
*   image multiplication operation giving a 16-bit product.
*
*   Image Processing Interface Library (IPI) call
*       developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*       IPI_IPILIB
*
*   Called Routines:
*
*       IPI_DVPMultiply
*       IPI_ErrorCheck
*
*   Modification History:
*
*   8/2/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.
 INTEGER*4 Unit ! IP8500 unit number.

INTEGER*2 Multiplicand_Channel ! 8-bit Multiplicand Image.
 INTEGER*2 Multiplier_Channel ! 8-bit Multiplier Image.
 INTEGER*2 Product_HI_Channel ! HI byte of product image.
 INTEGER*2 Product_LO_Channel ! LO byte of product image.

PARAMETER (Multiplicand_Channel = 0)
 PARAMETER (Multiplier_Channel = 1)
 PARAMETER (Product_HI_Channel = 2)
 PARAMETER (Product_LO_Channel = 3)

COMMON /IPI_Unit/ Unit ! IPI_Unit Common block.

```
Status = Ipi_DVPMultiply( Unit, Multiplicand_Channel,
                          Multiplier_Channel,
                          Product_HI_Channel,
                          Product_LO_Channel )
```

CALL Ipi_ErrorCheck(Status, 'Error Multiplying Channels')

RETURN

END ! DVP_Multiply_8bit

```

SUBROUTINE      DUP_NOOP_8bit( Operand1_Channel,
                               Result_Channel )

```

```

*****
*
*   This subroutine calls an IPI routine to execute an 8-bit DUP
*   image NOOP operation which has the effect of a copy operation
*   caused by a pass-thru mode.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_DUPMath
*       IPI_ErrorCheck
*
*   Modification History:
*
*   12/11/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INCLUDE 'IPI_IPILIB'           ! Include IPI header file.

```

```

INTEGER*4 Status                ! Status call variable.
INTEGER*4 Unit                  ! IP8500 unit number.
INTEGER*2 Operand1_Channel      ! First operand channel.
INTEGER*2 Result_Channel        ! Result channel.

```

```

COMMON /Ipi_Unit/ Unit          ! IPI_Unit common block.

```

```

Status = Ipi_DUPMath ( Unit,
.                   Ipi__NOP,
.                   Operand1_Channel,
.                   Result_Channel )

```

```

CALL Ipi_ErrorCheck( Status, 'Error NOOPing Channels' )

```

```

RETURN

```

```

END ! DUP_NOOP_8bit

```

```

SUBROUTINE      DVP_OR_8bit(      Operand1_Channel,
                                Operand2_Channel,
                                Result_Channel )

```

```

*****
*
*      This subroutine calls an IPI routine to execute an 8-bit DVP
*      image logical bitwise OR operation.
*
*      Image Processing Interface Library (IPI) call
*      developed by S. Schultz
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*              IPI_IPILIB
*
*      Called Routines:
*
*              IPI_DVPMath
*              IPI_ErrorCheck
*
*      Modification History:
*
*      12/11/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INCLUDE 'IPI_IPILIB'                ! Include IPI header file.

```

```

INTEGER*4 Status                    ! Status call variable.

```

```

INTEGER*4 Unit                      ! IP8500 unit number.

```

```

INTEGER*2 Operand1_Channel          ! First operand channel.

```

```

INTEGER*2 Operand2_Channel          ! Second operand channel.

```

```

INTEGER*2 Result_Channel            ! Result channel.

```

```

COMMON /Ipi_Unit/ Unit              ! IPI_Unit common block.

```

```

Status = Ipi_DVPMath ( Unit,
                        Ipi_OR,
                        Operand1_Channel,
                        Operand2_Channel,
                        Result_Channel )

```

```

CALL Ipi_ErrorCheck( Status, 'Error ORing Channels' )

```

```

RETURN

```

```

END ! DVP_OR_8bit

```

SUBROUTINE DVP_Subtract_16bit ! 16-bit signed subtraction

```
*****
*
*   This subroutine calls an IPI routine to execute a 16-bit DVP
*   image signed subtraction operation.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*       IPI_IPILIB
*
*   Called Routines:
*
*       IPI_DVPAdvMath
*       IPI_ErrorCheck
*
*   Modification History:
*
*   7/22/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

```
INTEGER*4 Status        ! Status call variable.
INTEGER*4 Unit        ! IP8500 unit number.
INTEGER*2 Minuend_HI_Channel    ! HI byte of minuend (BB).
INTEGER*2 Minuend_LO_Channel    ! LO byte of minuend (BA).
INTEGER*2 Subtrahend_HI_Channel ! HI byte of subtrahend (BD).
INTEGER*2 Subtrahend_LO_Channel ! LO byte of subtrahend (BC).
INTEGER*2 Difference_HI_Channel ! HI byte of difference (DVPB).
INTEGER*2 Difference_LO_Channel ! LO byte of difference (DVPA).
PARAMETER (Minuend_HI_Channel = 0)
PARAMETER (Minuend_LO_Channel = 1)
PARAMETER (Subtrahend_HI_Channel = 2)
PARAMETER (Subtrahend_LO_Channel = 3)
PARAMETER (Difference_HI_Channel = 2)
PARAMETER (Difference_LO_Channel = 3)
```

COMMON /IPI_Unit/ Unit ! IPI_Unit Common block.

```
Status = Ipi_DVPAdvMath( Unit, Ipi_Minus,
.                        Minuend_HI_Channel,
.                        Minuend_LO_Channel,
.                        Subtrahend_HI_Channel,
.                        Subtrahend_LO_Channel,
.                        Difference_HI_Channel,
.                        Difference_LO_Channel)
CALL Ipi_ErrorCheck( Status, 'Error Subtracting Channels')
```

RETURN

END ! DVP_Subtract_16bit


```

SUBROUTINE Enable_ITT( Channel,
                      VOC_ITT_Control,
                      DVP_ITT_Control )      ! ITT Controller.

```

```

*****
*
*   This subroutine clears individual 1024 x 1024 image channels
*   of the IP8500 by an IPI library call.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*       IPI_IPILIB
*
*   Called Routines:
*
*       IPI_ClearChan
*       IPI_ErrorCheck
*
*   Modification History:
*
*   4/5/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INCLUDE 'IPI_IPILIB'      ! Include IPI Header File.

INTEGER*4 Status           ! Status call variable.
INTEGER*4 Unit             ! IP8500 Unit Number.
INTEGER*4 Channel          ! ITT Channel number to be controlled

INTEGER*2 VOC_ITT_Control  ! Control Flag of VOC ITT.
INTEGER*2 DVP_ITT_Control  ! Control Flag of DVP ITT.

COMMON /Ipi_Unit/ Unit     ! Ipi_Unit common block.

Status = Ipi_EnableItt( Unit,
                       Channel,
                       VOC_ITT_Control,
                       DVP_ITT_Control )      ! ITT control call.

CALL Ipi_ErrorCheck( Status, 'Error Enabling/Disabling ITT')

RETURN

END ! Enable_ITT

```

[illegible]

[illegible]

```

                                y_position,
                                window_size);

                                break;

case 3:
    erode( bit_level,
           x_position,
           y_position,
           window_size / 2 );
    break;

default:
    printf("Error in Erosion");
}
}
}
}
}
/* End of erode function *****/

```


SUBROUTINE Erode_pixel(x, y) ! Erode the image by a single pixel.

```
*****
*
*   This subroutine erodes an image in channel 0 by a single
*   pixel of the structuring element found in channel 1. The
*   resulting intermediary contribution of this erosion is applied
*   to the result channel (channel 2).
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Scroll_Channel
*       Noop_8bit
*       Channel_Empty
*       And_8bit
*
*   Modification History:
*
*   9/27/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*2 x                ! structuring element pixel x-position.
INTEGER*2 y                ! structuring element pixel y-position.
INTEGER*4 Channel_mask     ! Bitwise channel selection variable.
INTEGER*2 x_scroll         ! x-scroll register variable.
INTEGER*2 y_scroll         ! y-scroll register variable
INTEGER*2 operand1_channel ! Channel number of first image operand
INTEGER*2 result_channel   ! Channel number of result image.
LOGICAL*1 first_image_scroll ! Flag test for first image scroll.
LOGICAL*1 channel_empty    ! Function testing for empty channel.
PARAMETER( operand1_channel = 0 ) ! First image in channel 0.
PARAMETER( result_channel = 2 ) ! Result image in channel 2.
PARAMETER( channel_mask = 2 ** operand1_channel ) ! Scroll channel 2.

x_scroll = -x + 256 ! Origin of unscrolled image is (256,256)
                  ! with x_scroll and y_scroll representing
                  ! the image coordinates of upper left corner
y_scroll = -( 512 - y ) + 767 ! after scroll.

CALL Scroll_channel( x_scroll, y_scroll, channel_mask ) ! Scroll call.

first_image_scroll = channel_empty( result_channel ) ! First scroll?

IF (first_image_scroll) THEN
    CALL Noop_8bit( operand1_channel, result_channel )
ELSE
    CALL and_8bit( operand1_channel, result_channel,
                  result_channel ) ! AND Image otherwise.
ENDIF

RETURN

END ! Erode_Pixel
```

```

/* Start of erode_quadrant_pixel function *****/
*
*   This function is used as a means to establish auto variables
*   which are referenced by the FORTRAN subroutine erode_square.
*   This is necessary in order to allow recursive functions to
*   call this routine and maintain unique parameter values stored
*   in the stack during recursive invocation of this function.
*   Without this function, recursive calls to erode_square cannot
*   be made because of FORTRAN's inherent limitation of passing all
*   parameters by reference, and its lack of dynamic storage
*   allocation.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       erode_square
*
*   Modification History:
*
*   9/27/90   Completed documentation & comments - Rolando Raqueño
*
*****/

/*-----
FUNCTION PROTOTYPE SECTION-----*/
extern void erode_square(      unsigned short int *ptr_x_position,
                              unsigned short int *ptr_y_position,
                              unsigned short int *ptr_window_size );

/*-----
ERODE_QUADRANT_PIXEL MAIN BODY-----*/
void erode_quadrant_pixel(      unsigned short int x_position,
                              unsigned short int y_position,
                              unsigned short int window_size )
{
    erode_square(      &x_position,
                      &y_position,
                      &window_size      );
}
/* End of erode_quadrant_pixel function *****/

```

```

subroutine erode_square(      x_lower_left_corner,
                             y_lower_left_corner,
                             window_size )

```

```

*****
*
*   This subroutine erodes an image in channel 0 by a square
*   section of the structuring element found in channel 1. The
*   resulting intermediary contribution of this erosion is applied
*   to the result channel (channel 2).
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*
*   Called Routines:
*       Erode_Pixel
*
*
*   Modification History:
*
*   9/27/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

implicit none

```

```

integer*2 x_lower_left_corner      ! x of lower left corner.
integer*2 y_lower_left_corner      ! y of lower left corner.
integer*2 window_size              ! size of square.

```

```

integer*2 x                        ! x coordinate of pixel.
integer*2 y                        ! y coordinate of pixel.

```

```

integer*2 x_upper_right_corner     ! x of upper right corner.
integer*2 y_upper_right_corner     ! y of upper right corner.

```

```

x_upper_right_corner = x_lower_left_corner + window_size - 1
y_upper_right_corner = y_lower_left_corner + window_size - 1

```

```

do x = x_lower_left_corner, x_upper_right_corner
  do y = y_lower_left_corner, y_upper_right_corner

```

```

    call erode_pixel( x, y )      ! Erode by a pixel

```

```

  end do

```

```

end do

```

```

return

```

```

end ! Erode_Square

```



```

REAL*4 FUNCTION Final_Corrected_Contour(Trace_File_Name,
                                         Vector_Channel,
                                         Corrected_Vector_Channel,
                                         Quad_tree_buffer,
                                         Window_size,
                                         x_neutral_axis,
                                         y_neutral_axis,
                                         Correction_Factor)

```

```

*
* This function operates in a similar manner to the Open_Contour
* except it operates on the final closed contour that has been
* morphologically processed. This function accepts a file name
* (Trace_File_Name) to which the contour trace data points will
* be written. Vector_Channel and Corrected_Vector_Channel gives
* the channels to which the morphologically processed contour and
* corrected contour will be displayed. The morphologically
* processed contour is the trace of the processed image. The
* corrected contour is the same trace as the morphologically
* processed contour except it has been expanded by some correc-
* tion factor from the centroid of the trace. This correction
* factor (Correction_Factor) is the net dimension by which the
* images were thickened and thinned by the dilation/erosion
* sequence.

```

```

*
* As an example, if an image were dilated by circular structuring
* element of diameter 91 pixels. This image would effectively be
* thickened isotropically by 45.5 pixels. A subsequent erosion
* with another structuring element of 45.5 pixels in diameter
* would thin the image by 22.75 pixels. Assuming that the image
* processed yields a closed cavity, the resulting correction
* factor at this point of the process will be 22.75. Therefore,
* the correction factor can be computed by taking the sum of the
* diameters of the the dilating structuring elements and
* subtracting the sum of the diameters of the eroding structuring
* elements and dividing the difference by two.

```

```

*
* The correction factor is applied by simply radially extending
* the distance of the morphologically processed trace from the
* centroid by the amount of the correction factor.

```

```

*
* Written by Rolando Raqueño (Center for Imaging Science)

```

```

*
* Include Files:

```

```

*
* Called Routines:

```

```

*
*       x_quadrant_position
*       y_quadrant_position
*       Pixel_Value_8bit
*       Area_under_vector
*       Draw_Vector
*       Clear_Channel

```

```

*
* Modification History:

```

```

*
* 9/16/90 Completed documentation & comments - Rolando Raqueño
* 9/30/90 Added call to Clear_Channel to clear
*         the contour trace drawn - Rolando Raqueño

```

IMPLICIT NONE

PARAMETER Definitions and Declarations.

```
INTEGER*2      Trace_File           ! Output file unit number.
INTEGER*2      Image_x_Size         ! x-size of image.
INTEGER*2      Image_y_Size         ! y-size of image.
INTEGER*2      Vector_Grey_Level    ! Grey level of Vector.
PARAMETER      (Trace_File = 1 )
PARAMETER      (Image_x_Size = 512 )
PARAMETER      (Image_y_Size = 512 )
PARAMETER      (Vector_Grey_Level = 255 )
```

Argument list Definitions

```
CHARACTER*80   Trace_File_Name      ! Trace output data file.
INTEGER*2      Vector_Channel        ! Channel to draw vector.
INTEGER*2      Corrected_Vector_Channel
BYTE           Quad_tree_buffer( Image_x_Size, Image_y_Size ) ! Array.
INTEGER*2      Window_Size           ! Quad-tree resolution.
INTEGER*2      x_neutral_axis        ! Search start x-position.
INTEGER*2      y_neutral_axis        ! Search start y-position.
INTEGER*2      Correction_Factor
```

Variable Definitions

```
INTEGER*2      Border_Value          ! Bit-plane Grey Value.
INTEGER*2      Starting_x_position    ! Image x-center point.
INTEGER*2      Starting_y_position    ! Image y-center point.
INTEGER*2      Center_pixel_value     ! Window center pixel.
INTEGER*2      Current_Window_x_position ! Window x-position.
INTEGER*2      Current_Window_y_position ! Window y-position.
INTEGER*2      Corrected_current_x_position
INTEGER*2      Corrected_current_y_position
INTEGER*2      first_border_x_position ! first border pixel x-pos.
INTEGER*2      first_border_y_position ! first border pixel y-pos.
INTEGER*2      window_x_position      ! x-position of search window.
INTEGER*2      window_y_position      ! y-position of search window.
INTEGER*2      neighbor_x_position     ! neighbor pixel x-position.
INTEGER*2      neighbor_y_position     ! neighbor pixel y-position.
INTEGER*2      next_window_x_position ! next border pixel x-position.
INTEGER*2      next_window_y_position ! next border pixel y-position.
INTEGER*2      Corrected_Next_x_position
INTEGER*2      Corrected_Next_y_position
INTEGER*2      neighbor_pixel_value    ! value of neighbor pixel.
INTEGER*2      delta_x                 ! delta x to neighbor pixel.
INTEGER*2      delta_y                 ! delta y to neighbor pixel.
INTEGER*2      search_angle            ! search angle for border.
LOGICAL*1      contour_area_is_open   ! Contour area is open flag.
INTEGER*2      Array_of_Points( 2,2 ) ! Start/End Coordinate Points.
INTEGER*2      Corrected_Points( 2,2 ) ! Corrected contour points
REAL*4         Correction_Delta_x      ! Correction Run
REAL*4         Correction_Delta_y      ! Correction Rise
REAL*4         Trace_Angle             ! Angle from centroid to point.
REAL*4         Area
```

Function Definitions

```
INTEGER*2      x_quadrant_position    ! x-position of super-pixel.
INTEGER*2      y_quadrant_position    ! y-position of super-pixel.
```

```

INTEGER*2      Pixel_Value_8bit      ! value of pixel at (x,y).
REAL*4         Area_Under_Vector      ! Calculates area under vector.

```

Equivalence Definitions

```

EQUIVALENCE ( Array_of_Points(1,1), Current_Window_x_position )
EQUIVALENCE ( Array_of_Points(2,1), Current_Window_y_position )
EQUIVALENCE ( Array_of_Points(1,2), Next_Window_x_position )
EQUIVALENCE ( Array_of_Points(2,2), Next_Window_y_position )
EQUIVALENCE ( Corrected_Points(1,1), corrected_current_x_position )
EQUIVALENCE ( Corrected_Points(2,1), corrected_current_y_position )
EQUIVALENCE ( Corrected_Points(1,2), Corrected_Next_x_position )
EQUIVALENCE ( Corrected_Points(2,2), Corrected_Next_y_position )

```

Start at the center of the image, determine quad-tree bit-plane to search, and start to find the first border pixel.

```

OPEN ( Unit = Trace_File, File = Trace_File_Name, Status = 'NEW' )

```

```

WRITE( Trace_File,*) x_neutral_axis, y_neutral_axis

```

```

Starting_x_position = x_quadrant_position( x_neutral_axis,
                                           Window_Size      ) ! start x.
Starting_y_position = y_quadrant_position( y_neutral_axis,
                                           Window_Size      ) ! start y.

```

```

Border_Value = 128 / Window_Size      ! Calculate Border value.

```

```

Area = 0.0                            ! Initialize Contour area.

```

```

Center_pixel_value = Pixel_Value_8bit( Quad_tree_buffer,
                                         Starting_x_position,
                                         Starting_y_position ) ! Value.

```

! If center pixel is a border pixel,
! then return open_contour = .FALSE.

```

IF ( Center_Pixel_Value .GE. Border_Value ) THEN

```

```

    TYPE *, ' Center pixel is a border pixel '

```

```

    RETURN

```

```

END IF

```

! Move left until until the first border pixel is found.

```

Current_Window_x_Position = Starting_x_position
Current_Window_y_Position = Starting_y_position

```

```

DO WHILE ( (Pixel_Value_8bit ( Quad_tree_buffer,
                               Current_Window_x_position,
                               Current_Window_y_position ))
           .LT. Border_Value )

```

```

    Current_Window_x_position = Current_Window_x_position -
                                Window_Size

```

```

END DO

```

After first pixel has been found start finding the next border pixel.

```

*      Start scanning the search window for the next border position.
*
First_border_x_position = Current_Window_x_position      ! 1st x border.
First_border_y_position = Current_Window_y_position      ! 1st y border.

Correction_Delta_x = Current_Window_x_position - x_neutral_axis
Correction_Delta_y = Current_Window_y_position - y_neutral_axis

Trace_Angle = atan2d( Correction_Delta_y, Correction_Delta_x )

corrected_current_x_position = Correction_Factor *
                                COSD( Trace_Angle ) +
                                Current_Window_x_position
corrected_current_y_position = Correction_Factor *
                                SIND( Trace_Angle ) +
                                Current_Window_y_position

contour_area_is_open = .TRUE.                                ! Assume open contour.
Search_angle = 0                                             ! Starting search angle

!Continue looking for border pixels while contour is open.
!Contour is considered closed when the first border pixel
!is encountered again.

DO WHILE ( contour_area_is_open )

    neighbor_pixel_value = 0                                ! Initialize variable.

    !The direction of each neighboring pixel with respect to
    !the current border pixel is given by a search angle.
    !
    !The next border pixel is found by examining the neighboring
    !pixels in a counter-clockwise order every 45°.
    !
    !Continue this search until a border pixel is found.

    DO WHILE( neighbor_pixel_value .LT. border_value)

        Search_angle = MOD( Search_angle + 45, 360 )

        Delta_x =      NINT( COSD( REAL( Search_angle ) ) ) *
                        Window_Size
        Delta_y =      NINT( SIND( REAL( Search_angle ) ) ) *
                        Window_Size

        neighbor_x_position = current_window_x_position +
                                delta_x
        neighbor_y_position = current_window_y_position +
                                delta_y

        neighbor_pixel_value = Pixel_Value_8bit(
                                Quad_tree_buffer,
                                neighbor_x_position,
                                neighbor_y_position )

    END DO

    !If a border pixel has been found, make a note of its
    !location, check to see if it is the same as the first
    !pixel, and move the search window to the new border pixel.
    !
    !With respect to the new position of the search window,

```



```
!advance the search angle for the next border pixel to start
!225 degrees counter-clockwise from the search angle made at
!the last search window position.
```

```
next_window_x_position = neighbor_x_position
next_window_y_position = neighbor_y_position
```

```
Correction_Delta_x = Next_Window_x_position - x_neutral_axis
Correction_Delta_y = Next_Window_y_position - y_neutral_axis
```

```
Trace_Angle = atan2d( Correction_Delta_y , Correction_Delta_x )
```

```
Corrected_next_x_position =      Correction_Factor *
                                COSD( Trace_Angle ) +
                                Next_Window_x_position
```

```
Corrected_next_y_position =      Correction_Factor *
                                SIND( Trace_Angle ) +
                                Next_Window_y_position
```

```
contour_area_is_open = .NOT. ( next_window_x_position
                                .EQ.
                                first_border_x_position
                                .AND.
                                next_window_y_position
                                .EQ.
                                first_border_y_position      )
```

```
CALL Draw_Vector(      Vector_Channel,
                        Vector_Grey_Level,
                        Array_of_Points )
CALL Draw_Vector(      Corrected_Vector_Channel,
                        Vector_Grey_Level,
                        Corrected_Points )
```

```
Area = Area + Area_under_vector( Corrected_Points )
```

```
WRITE( Trace_File, * ) Corrected_Current_x_position,
                        Corrected_Current_y_position
```

```
IF ( contour_area_is_open ) THEN
    Search_Angle = MOD( Search_Angle + 225, 360 )

    Current_Window_x_position = Next_window_x_position
    Current_Window_y_position = Next_window_y_position

    Corrected_Current_x_position = Corrected_Next_x_position
    Corrected_Current_y_position = Corrected_Next_y_position
```

```
END IF
```

```
END DO
```

```
CLOSE( Trace_File )
```

```
CALL Draw_Vector(      Corrected_Vector_Channel,
                        Vector_Grey_Level,
                        Corrected_Points )
```

```
Final_Corrected_Contour = Area
```

```
RETURN
```

```
END ! Final_Corrected_Contour.
```


SUBROUTINE Get_Histogram(Histogram_Buffer) ! Read the histogram.

```
*****
*
*   This subroutine reads current histogram results and load it
*   into a histogram buffer using an IPI library call.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_GetHst
*       IPI_ErrorCheck
*
*   Modification History:
*
*   4/9/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.

INTEGER*4 Unit ! IP8500 unit number.

INTEGER*4 Histogram_Buffer(0:255) ! 256 grey level Histogram buffer

COMMON /Ipi_Unit/ Unit ! Ipi_Unit common block.

Status = Ipi_GetHst(Unit, Histogram_Buffer) ! Get Histogram call.

CALL Ipi_ErrorCheck(Status, 'Error Getting Histogram')

RETURN

END ! Get_Histogram

SUBROUTINE Go_DUP ! Initiate DUP operation.

```
*****
*
*   This subroutine signals the DUP to initiate an operation
*   based on previous DUP control calls.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_DUPGo
*
*   Modification History:
*
*   7/20/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB'

INTEGER*4 Status ! Status call variable.
INTEGER*4 Unit ! IP8500 Unit number.

COMMON /IPI_Unit/ Unit ! IPI_Unit Common block.

Status = Ipi_DUPGo (Unit)
CALL Ipi_ErrorCheck(Status, 'Error Going DUP')

RETURN ! GO_DUP

END

SUBROUTINE GRAD(INPUT_FILE)

```
*****
*
*   This subroutine calculates the GRAD image of a given input file
*   using the DVP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       GRAD_0_Degree
*       GRAD_45_Degree
*       GRAD_90_Degree
*       GRAD_135_Degree
*       Maximum_Magnitudes_16bit
*
*   Modification History:
*
*   8/1/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

CHARACTER*80 INPUT_FILE

```
CALL GRAD_0_Degree( Input_File )
CALL GRAD_45_Degree( Input_File )
CALL GRAD_90_Degree( Input_File )
CALL GRAD_135_Degree( Input_File )
```

CALL Maximum_Magnitudes_16bit

RETURN

END ! GRAD

SUBROUTINE GRAD_0_DEGREE(INPUT_FILE)

```
*****
*
*   This subroutine calculates a 16-bit derivative of the kernel
*
*           -1  0  1
*
*   and calculates the magnitude of the derivative using the DUP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*           Derivative_0_Degree_16bit
*           Magnitude_0_Degree_16bit
*
*   Modification History:
*
*   8/1/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

CHARACTER*80 INPUT_FILE

CALL DERIVATIVE_0_DEGREE_16BIT(INPUT_FILE)

CALL MAGNITUDE_0_DEGREE_16BIT

RETURN

END ! GRAD_0_Degree

SUBROUTINE GRAD_45_DEGREE(INPUT_FILE)

```
*****
*
*   This subroutine calculates a 16-bit derivative of the kernel
*
*           -1
*           0
*           1
*
*   and calculates the magnitude of the derivative using the DUP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Derivative_45_Degree_16bit
*       Magnitude_45_Degree_16bit
*
*   Modification History:
*
*   8/1/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

CHARACTER*80 INPUT_FILE

CALL DERIVATIVE_45_DEGREE_16BIT(INPUT_FILE)

CALL MAGNITUDE_45_DEGREE_16BIT

RETURN

END ! GRAD_45_Degree

SUBROUTINE GRAD_90_DEGREE(INPUT_FILE)

```
*****
*
*   This subroutine calculates a 16-bit derivative of the kernel
*
*               -1
*               0
*               1
*
*   and calculates the magnitude of the derivative using the DUP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Derivative_90_Degree_16bit
*       Magnitude_90_Degree_16bit
*
*   Modification History:
*
*   8/1/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

CHARACTER*80 INPUT_FILE

CALL DERIVATIVE_90_DEGREE_16BIT(INPUT_FILE)

CALL MAGNITUDE_90_DEGREE_16BIT

RETURN

END ! GRAD_90_Degree

SUBROUTINE Grey_Grad(INPUT_FILE)

```
*****
*
*   This subroutine calculates the Grey_Grad image for a given
*   input file by loading the Grey image in channel 0 and the
*   Grad image in channel 1 and multiplying the two 8-bit images
*   to create a 16-bit product image using the DVP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Load_Grad_Image
*       Load_Grey_Image
*       Multiply_8bit
*       Store_Result_16bit
*
*   Modification History:
*
*   8/1/90  Completed documentation & comments - Rolando Raqueño
*   8/3/90  Added Store_Result_16bit routine - Rolando Raqueño
*
*****
```

IMPLICIT NONE

CHARACTER*80 INPUT_FILE
CHARACTER*80 Output_File

Output_File = 'Grey_Grad.Image'

CALL Load_Grey_Image(Input_File) ! Multiplicand in channel 0.
CALL Load_Grad_Image ! Multiplier in channel 1.
CALL Multiply_8bit ! 16-bit product image.
CALL Store_Result_16bit(Output_File) ! Save result in channel(2,3).

RETURN

END ! Grey_GRAD

SUBROUTINE Histogram_Channel(Histogram_Buffer, Channel)

```

*****
*
*   This subroutine calculates the histogram distribution for a
*   full rectangular region of interest in a specific channel.
*   The histogram distribution result is stored in the array
*   Histogram_Buffer.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Attach_DUP
*       Initialize_DUP
*       Clear_Histogram
*       Calculate_Histogram
*       Get_Histogram
*       Detach_DUP
*
*   Modification History:
*
*   8/3/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INTEGER*4 Histogram_Buffer( 0:255 )    ! 256 grey level buffer.
INTEGER*4 Channel                      ! Channel to histogram.

CALL Attach_DUP                        ! Get DUP for use.
CALL Initialize_DUP                    ! Initialize the DUP.
CALL Clear_Histogram                   ! Clear hardware buffer.
CALL Calculate_Histogram( Channel )    ! Compute histogram.
CALL Get_Histogram( Histogram_Buffer ) ! Get buffer from hardware.
CALL Detach_DUP                        ! Release DUP resource.

```

RETURN

END ! Histogram_Channel

```

SUBROUTINE Histogram_Channel_ROI(      Histogram_Buffer,
.                                     Channel,
.                                     X_LLC,
.                                     Y_LLC,
.                                     X_URC,
.                                     Y_URC )

```

```

*****
*
*   This subroutine calculates the histogram distribution for a
*   given rectangular region of interest in a specific channel.
*   The region of interest is defined by the x and y coordinates
*   of a lower left corner point and an upper right corner point.
*   The histogram distribution result is stored in the array
*   Histogram_Buffer.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Attach_DVP
*       Initialize_DVP
*       Clear_Histogram
*       Region_of_Interest
*       Calculate_Histogram_ROI
*       Get_Histogram
*       Detach_DVP
*
*   Modification History:
*
*   5/31/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INTEGER*4 Histogram_Buffer( 0:255 )      ! 256 Grey-level Histogram.
INTEGER*4 Channel                        ! Channel to histogram.

INTEGER*2 X_LLC                         ! X lower left corner.
INTEGER*2 Y_LLC                         ! Y lower left corner.
INTEGER*2 X_URC                         ! X upper right corner.
INTEGER*2 Y_URC                         ! Y upper right corner.

CALL Attach_DVP                          ! Attach DVP.
CALL Initialize_DVP                      ! Initialize DVP.
CALL Clear_Histogram                    ! Clear histogram buffer.
CALL Region_of_Interest( X_LLC, Y_LLC, X_URC, Y_URC ) ! Define ROI
CALL Calculate_Histogram_ROI( Channel ) ! Histogram channel.
CALL Get_Histogram( Histogram_Buffer ) ! Read histogram buffer.
CALL Detach_DVP                         ! Detach DVP.

```

RETURN

END ! Histogram_Channel_ROI

INTEGER*4 FUNCTION Histogram_High(Histogram_Buffer)

```
*****
*
*      This function calls an IPI library routine to calculate
*      histogram statistics and determine the highest grey level
*      tallied in the histogram buffer.
*
*      Image Processing Interface Library (IPI) call
*      developed by S. Schultz
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*              IPI_IPILIB
*
*      Called Routines:
*
*              IPI_StatHst
*              IPI_ErrorCheck
*
*      Modification History:
*
*      12/13/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INTEGER*4 Status	! Status call variable.
INTEGER*4 Low	! Lowest grey value tallied.
INTEGER*4 High	! Highest grey value tallied.
INTEGER*4 Maximum	! Mode.
INTEGER*4 Total	! Total pixels tallied.
INTEGER*4 Histogram_Buffer(0:255)	! 256 Grey-level histogram.

Status = Ipi_StatHst(Histogram_Buffer, , Low, High, Maximum, Total)
Call Ipi_ErrorCheck(Status, 'Error Calculating Histogram Statistics')

Histogram_high = High ! Return highest grey level.

RETURN

END ! Histogram_High

INTEGER*4 FUNCTION Histogram_Low(Histogram_Buffer)

```
*
*      This function calls an IPI library routine to calculate
*      histogram statistics and determine the highest grey level
*      tallied in the histogram buffer.
*
*      Image Processing Interface Library (IPI) call
*      developed by S. Schultz
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*          IPI_IPILIB
*
*      Called Routines:
*          IPI_StatHst
*          IPI_ErrorCheck
*
*      Modification History:
*
*      12/13/90  Completed documentation & comments - Rolando Raqueño
*
```

IMPLICIT NONE

```
INTEGER*4 Status           ! Status call variable.
INTEGER*4 Low              ! Lowest grey value tallied.
INTEGER*4 High             ! Highest grey value tallied.
INTEGER*4 Maximum          ! Mode.
INTEGER*4 Total            ! Total pixels tallied.
INTEGER*4 Histogram_Buffer( 0:255 ) ! 256 Grey-level Histogram.
```

```
Status = Ipi_StatHst( Histogram_Buffer, , Low, High, Maximum, Total )
Call Ipi_ErrorCheck( Status, 'Error Calculating Histogram Statistics' )
```

```
Histogram_Low = Low           ! Return lowest grey level
```

RETURN

END ! Histogram_Low

INTEGER*4 FUNCTION Histogram_Maximum(Histogram_Buffer)

```
*****
*
*   This function calls an IPI library routine to calculate
*   histogram statistics and determine the mode in the histogram.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_StatHst
*       IPI_ErrorCheck
*
*   Modification History:
*
*   12/13/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INTEGER*4 Status	! Status call variable.
INTEGER*4 Low	! Lowest grey value tallied.
INTEGER*4 High	! Highest grey value tallied.
INTEGER*4 Maximum	! Mode.
INTEGER*4 Total	! Total pixels tallied.
INTEGER*4 Histogram_Buffer(0:255)	! 256 Grey-level Histogram.

Status = Ipi_StatHst(Histogram_Buffer, , Low, High, Maximum, Total)
Call Ipi_ErrorCheck(Status, 'Error Calculating Histogram Statistics')

Histogram_Maximum = Maximum ! Return mode.

RETURN

END ! Histogram_maximum

INTEGER*4 FUNCTION Histogram_Total(Histogram_Buffer)

```
*****
*
*   This function calls an IPI library routine to calculate
*   histogram statistics and determine the total number of
*   pixels tallied in the histogram buffer.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_StatHst
*       IPI_ErrorCheck
*
*   Modification History:
*
*   5/31/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Status           ! Status call variable.
INTEGER*4 Low              ! Lowest grey value tallied.
INTEGER*4 High             ! Highest grey value tallied.
INTEGER*4 Maximum          ! Mode.
INTEGER*4 Total            ! Total pixels tallied.
INTEGER*4 Histogram_Buffer( 0:255 ) ! 256 Grey-level Histogram.
```

Status = Ipi_StatHst(Histogram_Buffer, , Low, High, Maximum, Total)

Call Ipi_ErrorCheck(Status, 'Error Calculating Histogram Statistics')

Histogram_total = Total ! Return total pixels tallied.

RETURN

END ! Histogram_Total

SUBROUTINE

Initialize_IP

!Initialize IP8500

```
*****
*
*   This subroutine is similar to Initialize_IP_no_clear except
*   all the channels and sectors are cleared.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Initialize_IP_no_clear
*       Clear_Channels_Extended
*
*   Modification History:
*
*   4/9/90 Completed documentation & comments - Rolando Raqueño
*   12/11/90 Replaced individual calls with call to
*           initialize_ip_no_clear - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
CALL Initialize_ip_no_clear    ! Attach and initialize IP.
CALL Clear_Channels_Extended  ! Clear all channels and sectors.
```

RETURN

END ! Initialize_Ip

SUBROUTINE Initialize_IP_No_Clear ! Initialize IP8500 w/o clear.

```
*****
*
*   This subroutine initializes the IP8500 system by
*
*   1) Attaching the image processing system.
*   2) Setting the picture settings to a standard monochrome image.
*   3) Clearing the scroll registers.
*   4) Disabling all the Intensity Transformation Tables (ITT).
*   5) Resets the viewing mode to normal RGB viewing of channels.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Attach_IP
*       Set_Mono_Picture_Options
*       Clear_Scroll
*       Disable_ITTs
*       Reset_View_Channel
*
*   Modification History:
*
*   4/5/90  Completed documentation & comments - Rolando Raqueño
*   10/2/90 Added call to Reset_View_Channel - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
CALL Attach_IP           ! Attach IP8500.
CALL Set_Mono_Picture_Options ! Standard monochrome image.
CALL Clear_Scroll        ! Reset scroll to normal settings.
CALL Disable_ITTs        ! Disable ITT's
CALL Reset_View_Channel   ! Reset viewing to normal RGB.
```

RETURN

END ! Initialize_IP_No_Clear

SUBROUTINE Initialize_DUP ! Initialize DUP to pass-thru mode

```
*****
*
*   This subroutine initializes the DUP to a pass-thru mode by an
*   IPI library call.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*           IPI_IPILIB
*
*   Called Routines:
*           IPI_DUPInit
*           IPI_ErrorCheck
*
*   Modification History:
*
*   4/9/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.

INTEGER*4 Unit ! IP8500 unit number.

COMMON /Ipi_Unit/ Unit ! IPI_Unit common block.

Status = Ipi_DUPInit(Unit) ! Initialize DUP call.

CALL Ipi_ErrorCheck(Status, 'Error Initializing DUP')

RETURN

END ! Initialize_DUP.

SUBROUTINE Load_First_Operand_16Bit(File_Name)

```
*
*      This subroutine loads a 16-bit image as a 16-bit operand
*      into channel (0,1).
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*
*      Include Files:
*
*
*      Called Routines:
*          Clear_Scroll
*          Set_Word_Picture_Options
*          Show_Picture_16bit
*
*
*      Modification History:
*
*      4/9/90 Completed documentation & comments - Rolando Raqueño
*      7/22/90 Added Clear_Channels call - Rolando Raqueño
*      7/24/90 Modified to load operands in (0,1) - Rolando Raqueño
*      8/7/90 Swapped Channel Mask values to conform with DEC
*             HI and LO order byte convention - Rolando Raqueño
*
```

IMPLICIT NONE

```
CHARACTER*80 File_Name           ! 16-bit Image file name.
INTEGER*4 Channel_Mask(0:1)      ! Bitwise channel selection.

DATA Channel_Mask /2,1/          ! Select channels (0,1)

CALL Clear_Scroll                 ! Clear Scroll (All channels).
CALL Set_Word_Picture_Options     ! Specify image as 16-bits.
CALL Show_Picture_16bit( File_Name, Channel_Mask ) ! Load Images
```

RETURN

END ! Load_First_Operand_16bit

```

SUBROUTINE      Load_Grad_Image      ! Load channel 1.

*****
*
*      This subroutine loads the 16-bit grad image as an 8-bit operand
*      into channel 1.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*          Load_Lo8_of_16bit_to_Channel_1
*
*      Modification History:
*
*      8/3/90  Completed documentation & comments - Rolando Raqueño
*
*****

IMPLICIT NONE

CHARACTER*80 File_Name      ! 16-bit Image file name.

File_Name = 'Grad.Image'
CALL Load_Lo8_of_16bit_to_Channel_1( File_Name )

RETURN

END ! Load_Grad_Image

```

SUBROUTINE Load_Grey_Grad_Image ! Load Grey-Grad.

```
*
*      This subroutine loads the 16-bit Grey-Grad value image into
*      channel (2,3).
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*          Load_Second_Operand_16bit
*
*      Modification History:
*
*      8/5/90 Completed documentation & comments - Rolando Raqueño
*
```

IMPLICIT NONE

CHARACTER*80 File_Name ! Derivative Input Image.

File_Name = 'Grey_Grad.Image'

CALL Load_Second_Operand_16bit(File_Name) ! into channel (2,3).

RETURN

END ! Load_Grey_Grad_Image


```

SUBROUTINE      Load_Grey_Image( File_Name )      ! Load channel 0.

*****
*
*      This subroutine loads the 16-bit grey image as an 8-bit operand
*      into channel 0.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*              Load_Lo8_of_16bit_to_Channel_0
*
*      Modification History:
*
*      8/2/90  Completed documentation & comments - Rolando Raqueño
*
*****

      IMPLICIT NONE

      CHARACTER*80 File_Name                      ! 8-bit Image file name.

      CALL Load_Lo8_of_16bit_to_Channel_0( File_Name )

      RETURN

      END ! Load_Grey_Image

```

SUBROUTINE Load_Lo8_of_16bit_to_channel_0(File_Name)

```
*
*      This subroutine loads the lower 8-bits of a 16-bit image
*      into channel 0 (used when the upper 8-bits are empty).
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*
*      Include Files:
*
*
*      Called Routines:
*          Clear_Scroll
*          Set_Word_Picture_Options
*          Show_Picture_16bit
*
*
*      Modification History:
*
*      8/3/90  Completed documentation & comments - Rolando Raqueño
*      8/7/90  Swapped Channel Mask values to conform with DEC
*              HI and LO order byte convention - Rolando Raqueño
*
```

IMPLICIT NONE

```
CHARACTER*80 File_Name           ! 16-bit Image file name.
INTEGER*4 Channel_Mask(0:1)      ! Bitwise channel selection.

DATA Channel_Mask /1,0/          ! Channel 0 for low 8 bits.

CALL Clear_Scroll                 ! Clear Scroll (All channels).
CALL Set_Word_Picture_Options     ! Specify image as 16-bits.
CALL Show_Picture_16bit( File_Name, Channel_Mask ) ! Load Images.
```

RETURN

END ! Load_Lo8_of_16bit_to_Channel_0

SUBROUTINE Load_Lo8_of_16bit_to_channel_1(File_Name)

```
*****
*
* This subroutine loads the lower 8-bits of a 16-bit image
* into channel 1 (used when the upper 8-bits are empty).
*
* Written by Rolando Raqueño (Center for Imaging Science)
*
* Include Files:
*
* Called Routines:
*      Clear_Scroll
*      Set_Word_Picture_Options
*      Show_Picture_16bit
*
* Modification History:
*
* 8/3/90 Completed documentation & comments - Rolando Raqueño
* 8/7/90 Swapped Channel Mask values to conform with DEC
*      HI and LO order byte convention - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
CHARACTER*80 File_Name           ! 16-bit Image file name.
INTEGER*4 Channel_Mask(0:1)      ! Bitwise channel selection.

DATA Channel_Mask /2,0/          ! Channel 1 for low 8 bits.

CALL Clear_Scroll                 ! Clear Scroll (All channels)
CALL Set_Word_Picture_Options     ! Specify image as 16-bits.
CALL Show_Picture_16bit( File_Name, Channel_Mask ) ! Load Images
```

RETURN

END ! Load_Lo8_of_16bit_to_Channel_1

SUBROUTINE Load_Second_Operand_16Bit(File_Name)

```
*****
*
*   This subroutine loads a 16-bit image as a 16-bit operand
*   into channel (2,3).
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       Clear_Scroll
*       Set_Word_Picture_Options
*       Show_Picture_16bit
*
*   Modification History:
*
*   4/9/90  Completed documentation & comments - Rolando Raqueño
*   7/22/90 Added Clear_Channels call - Rolando Raqueño
*   7/24/90 Modified to load operands in (2,3) - Rolando Raqueño
*   8/7/90  Swapped Channel Mask values to conform with DEC
*           H1 and L0 order byte convention - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

CHARACTER*80 File_Name ! 8-bit Image file name.

INTEGER*4 Channel_Mask(2:3) ! Bitwise channel selection.

DATA Channel_Mask /8,4/ ! Select channels (2,3)

CALL Clear_Scroll ! Clear Scroll (All channels).

CALL Set_Word_Picture_Options ! Specify image as 16-bits.

CALL Show_Picture_16bit(File_Name, Channel_Mask) ! Load Images.

RETURN

END ! Load_Second_Operand_16bit

SUBROUTINE Magnitude_0_Degree_16bit ! Absolute Value Image

```

*****
*
*   This subroutine calculates a 16-bit magnitude of the
*   derivative of the kernel
*
*               -1  0  1
*
*   using the DUP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channels
*       Load_Second_Operand_16bit
*       Subtract_16bit
*       Load_First_Operand_16bit
*       Maximum_16bit
*       Store_Result_16bit
*
*   Modification History:
*
*   7/30/90 Completed documentation & comments - Rolando Raqueño
*   8/2/90  Used Store_Result_16bit routine - Rolando Raqueño
*   8/7/90  Removed redundant call to Clear_Scroll routine
*           - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80   Input_File           ! Derivative Input Image.
CHARACTER*80   Output_File          ! Magnitude Output Image.

Input_File = 'Derivative.0'         ! Name of input file.
Output_File = 'Magnitude.0'         ! Name of output file.

CALL Clear_Channels                  ! Clear all channels.
CALL Load_Second_Operand_16bit( Input_File ) ! Load Image.
CALL Subtract_16bit                  ! Get 2's complement.
CALL Load_First_Operand_16bit( Input_File ) ! Load Original Image.
CALL Maximum_16bit                   ! Find Absolute Value Image.
CALL Store_Result_16bit( Output_File ) ! Save intermediate result.

RETURN

END ! Magnitude_0_Degree_16bit

```


SUBROUTINE Magnitude_90_Degree_16bit ! Absolute Value Image

```

*****
*
*   This subroutine calculates a 16-bit magnitude of the
*   derivative of the kernel
*
*           -1
*           0
*           1
*
*   using the DUP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channels
*       Load_Second_Operand_16bit
*       Subtract_16bit
*       Load_First_Operand_16bit
*       Maximum_16bit
*       Store_Result_16bit
*
*   Modification History:
*
*   7/30/90 Completed documentation & comments - Rolando Raqueño
*   8/3/90  Used Store_Result_16bit routine - Rolando Raqueño
*   8/7/90  Removed redundant call to Clear_Scroll routine
*           - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80   Input_File           ! Derivative Input Image.
CHARACTER*80   Output_File          ! Magnitude Output Image.

Input_File = 'Derivative.90'        ! Name of input file.
Output_File = 'Magnitude.90'        ! Name of output file.

CALL Clear_Channels                  ! Clear all channels.
CALL Load_Second_Operand_16bit( Input_File ) ! Load Image.
CALL Subtract_16bit                  ! Get 2's complement.
CALL Load_First_Operand_16bit( Input_File ) ! Load Original Image.
CALL Maximum_16bit                   ! Find Absolute Value Image.
CALL Store_Result_16bit( Output_File ) ! Save intermediate result.

RETURN

END ! Magnitude_90_Degree_16bit

```

SUBROUTINE Magnitude_135_Degree_16bit ! Absolute Value Image.

```

*****
*
*   This subroutine calculates a 16-bit magnitude of the
*   derivative of the kernel
*
*
*           -1
*          0
*         1
*
*   using the DUP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Channels
*       Load_Second_Operand_16bit
*       Subtract_16bit
*       Load_First_Operand_16bit
*       Maximum_16bit
*       Store_Result_16bit
*
*   Modification History:
*
*   7/30/90 Completed documentation & comments - Rolando Raqueño
*   8/3/90  Used Store_Result_16bit routine - Rolando Raqueño
*   8/7/90  Removed redundant call to Clear_Scroll routine
*           - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80   Input_File           ! Derivative Input Image.
CHARACTER*80   Output_File          ! Magnitude Output Image.

```

```

Input_File = 'Derivative.135'      ! Name of input file.
Output_File = 'Magnitude.135'      ! Name of output file.

```

```

CALL Clear_Channels                ! Clear all channels.
CALL Load_Second_Operand_16bit( Input_File ) ! Load Image.
CALL Subtract_16bit                ! Get 2's complement.
CALL Load_First_Operand_16bit( Input_File ) ! Load Original Image.
CALL Maximum_16bit                 ! Find Absolute Value Image.
CALL Store_Result_16bit( Output_File ) ! Save intermediate result.

```

RETURN

END ! Magnitude_135_Degree_16bit

SUBROUTINE Maximum_16bit ! Pixel-by-pixel 16-bit Maximum.

```
*****
*
*   This subroutine calculates a 16-bit pixel-by-pixel maximum
*   using the DVP.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Attach_DVP
*       Initialize_DVP
*       Setup_MinMax_DVP_Inputs_16bit
*       Setup_MinMax_DVP_Outputs_16bit
*       DVP_Maximum_16bit
*       GO_DVP
*       Detach_DVP
*
*   Modification History:
*
*   7/20/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
CALL Attach_DVP
CALL Initialize_DVP
CALL Setup_MinMax_DVP_Inputs_16bit
CALL Setup_MinMax_DVP_Outputs_16bit
CALL DVP_Maximum_16bit
CALL GO_DVP
CALL Detach_DVP
```

RETURN

END ! Maximum_16bit

SUBROUTINE Maximum_Magnitudes_16bit ! Maximum Magnitudes.

```
*****
*
*   This subroutine calculates a 16-bit maximum magnitude of all
*   the 0°, 45°, 90°, and 135° derivatives.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Load_First_Operand_16bit
*       Load_Second_Operand_16bit
*       Maximum_16bit
*       Store_Result_16bit
*
*   Modification History:
*
*   7/30/90 Completed documentation & comments - Rolando Raqueño
*   8/7/90  Removed redundant calls to Clear_Scroll and
*           Clear_Channels routines - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
CHARACTER*80    Input_File                    ! Derivative Input Image.
CHARACTER*80    Output_File                  ! Magnitude Output Image.

Output_File = 'Grad.Image'                  ! Name of output file.

Input_File = 'Magnitude.0'                  ! 0° Derivative Magnitude
CALL Load_First_Operand_16bit( Input_File )    ! into channel (0,1).

Input_File = 'Magnitude.45'                  ! 45° Derivative Magnitude
CALL Load_Second_Operand_16bit( Input_File )    ! into channel (2,3).
CALL Maximum_16bit                          ! Find Maximum Magnitude.

Input_File = 'Magnitude.90'                  ! 90° Derivative Magnitude
CALL Load_First_Operand_16bit( Input_File )    ! into channel (0,1).
CALL Maximum_16bit                          ! Find Maximum Magnitude.

Input_file = 'Magnitude.135'                  ! 135° Derivative Magnitude
CALL Load_First_Operand_16bit( Input_File )    ! into channel (0,1).
CALL Maximum_16bit                          ! Find Maximum Magnitude.

CALL Store_Result_16bit( Output_File )    ! Save intermediate result.

RETURN

END ! Maximum_Magnitudes_16bit
```

```

SUBROUTINE move_channel_to_8bit_buffer( Channel_Number,
                                         Buffer_512x512x8bits )

```

```

*****
*
*   This subroutine moves an 8-bit image from a channel into
*   an 8-bit memory buffer.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Set_Mono_Picture_Options
*       Move_Channel_8bit_to_Buffer
*
*   Modification History:
*
*   9/10/90  Completed documentation & comments - Rolando Raqueño
*   9/29/90  Modified to take any channel number - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INTEGER*2 Channel_Number
BYTE Buffer_512x512x8bits( 0:511 , 0:511 )      ! Buffer.
INTEGER*4 Channel_Mask                          ! Bitwise Channel Selection.

Channel_Mask = 2 ** Channel_Number              ! Select channel.

CALL Set_Mono_Picture_Options                    ! Specify Image as 8-bits.
CALL Set_y_axis_inverted                        ! Align Screen/Buffer (x,y).

CALL Move_Channel_8bit_to_Buffer(                Buffer_512x512x8bits,
                                                Channel_Mask      )

RETURN

END ! move_channel_to_8bit_buffer

```

```

SUBROUTINE Move_Channel_8bit_to_Buffer( Buffer_512x512x8bits,
                                         Channel_Mask
                                         )

```

```

*****
*
*   This subroutine moves an 8-bit image from one of the IP8500's
*   memory channels into an 8-bit buffer as specified by previously
*   Picture Options.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*       IPI_IPILIB
*
*   Called Routines:
*
*       IPI_GetPic
*       IPI_ErrorCheck
*
*   Modification History:
*
*   9/10/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INCLUDE 'IPI_IPILIB'

```

```

INTEGER*4      Status      ! Status call variable.
INTEGER*4      Unit        ! IP8500 unit number.
INTEGER*4      PicOps      ! Picture Option settings.
BYTE           Buffer_512x512x8bits( 0:511 , 0:511 ) ! Buffer.
INTEGER*4      Channel_Mask ! Bitwise Channel selection.

```

```

COMMON /IPI_Unit/ Unit        ! IPI_Unit common block.
COMMON /IPI_PicOps/ PicOps    ! IPI_PicOps common block.

```

```

Status = Ipi_GetPic( Unit,
                    Channel_Mask,
                    Buffer_512x512x8bits,
                    PicOps )

```

```

CALL Ipi_ErrorCheck( Status, 'Error Moving Channel(s) to Buffer' )

```

```

RETURN

```

```

END ! Move_Channel_8bit_to_Buffer

```



```

SUBROUTINE Move_Channel_16bit_to_Buffer( Buffer_512x512x16bits,
                                         Channel_Mask
                                         )

```

```

*****
*
*   This subroutine moves a 16-bit image from two of the IP8500's
*   memory channels into a 16-bit buffer as specified by previously
*   Picture Options.
*
*   Image Processing Interface Library (IPI) call
*       developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_GetPic
*       IPI_ErrorCheck
*
*   Modification History:
*
*   8/6/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INCLUDE 'IPI_IPILIB'

```

```

INTEGER*4 Status           ! Status call variable.
INTEGER*4 Unit             ! IP8500 unit number.
INTEGER*4 PicOps           ! Picture Option settings.
INTEGER*2 Buffer_512x512x16bits( 0:511 , 0:511 ) ! Buffer.
INTEGER*4 Channel_Mask( 2 ) ! Bitwise Channel selection.

```

```

COMMON /IPI_Unit/ Unit      ! IPI_Unit common block.
COMMON /IPI_PicOps/ PicOps  ! IPI_PicOps common block.

```

```

Status = Ipi_GetPic( Unit,
                    Channel_Mask,
                    Buffer_512x512x16bits,
                    PicOps )

```

```

CALL Ipi_ErrorCheck( Status, 'Error Moving Channel(s) to Buffer' )

```

```

RETURN

```

```

END ! Move_Channel_16bit_to_Buffer

```

SUBROUTINE Move_Operand2_16bit_to_Buffer(Buffer_512x512x16bits)

```
*
*      This subroutine moves a 16-bit image from channel (2,3) into
*      a 16-bit memory buffer.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*          Set_Word_Picture_Options
*          Move_Channel_16bit_to_Buffer
*
*      Modification History:
*
*      8/6/90  Completed documentation & comments - Rolando Raqueño
*      8/7/90  Swapped Channel Mask values to conform with DEC
*              HI and LO order byte convention - Rolando Raqueño
*
```

IMPLICIT NONE

```
INTEGER*2 Buffer_512x512x16bits( 0:511 , 0:511 )      ! Buffer.
INTEGER*4 Channel_Mask( 2:3 )      ! Bitwise Channel Selection.
```

```
DATA Channel_Mask /8,4/      ! Select channel (2,3).
```

```
CALL Set_Word_Picture_Options      ! Specify Image as 16-bits.
CALL Move_Channel_16bit_to_Buffer( Buffer_512x512x16bits,
                                   Channel_Mask      )
```

RETURN

END ! Move_Operand2_16bit_to_Buffer

SUBROUTINE Multiply_8bit ! 8-bit multiplication.

```
*****
*
*   This subroutine calculates the product of two 8-bit images
*   using the DVP. The multiplicand image will be in channel 0
*   and the multiplier image will be in channel 1. The resulting
*   16-bit product image will be deposited in channels (2,3).
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Attach_DVP
*       Initialize_DVP
*       DVP_Multiply_8bit
*       Detach_DVP
*
*   Modification History:
*
*   8/1/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
CALL Attach_DVP           ! Get the DVP for use.
CALL Initialize_DVP       ! Initialize the DVP.
CALL DVP_Multiply_8bit    ! Start DVP multiply Operation
CALL Detach_DVP           ! Release DVP resource.
```

RETURN

END ! Multiply_8bit

```

SUBROUTINE      Noop_8bit(      Operand_Channel,
                             Result_Channel ) ! Image NOOP.

```

```

*****
*
*   The following subroutine performs an image NOOP DUP operation
*   in which the image in Operand_Channel is passed through the
*   DUP unaltered and placed in Result_Channel.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Attach_DUP
*       Initialize_DUP
*       DUP_NOOP_8bit
*       Detach_DUP
*
*   Modification History:
*
*   4/6/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INTEGER*2 Operand_Channel      ! Channel Number of Image Source.
INTEGER*2 Result_Channel      ! Channel Number of Image Sink.

```

```

CALL Attach_DUP                ! Allocate the DUP.
CALL Initialize_DUP            ! Initialize and Reset the DUP.

```

```

CALL DUP_Noop_8bit(      Operand_Channel,
                        Result_Channel)      ! NOOP Operation.

```

```

CALL Detach_DUP                ! Deallocate the DUP.

```

```

RETURN

```

```

END ! NOOP_8Bit.

```



```

REAL*4 FUNCTION Open_Contour(
.
.
.
.
.
.
Vector_Channel,
Quad_tree_buffer,
Window_size,
first_x_position,
first_y_position,
x_neutral_axis,
y_neutral_axis )

```

This function determines whether or not a closed contour generates a negative (hollow) or positive (solid) area for a given resolution in the 8-bit quad-tree. The resolution level to search is determined by the window size parameter.

The routine starts in the center of the image and scans a line across the image to the left until the first border pixel is found. The search then continues using a search window.

The contour following process uses a 3x3 search window of the appropriate resolution to determine the location of the next adjacent border pixel.

3	2	1
4	Current Border Pixel	0
5	6	7

The search of the neighboring pixels proceeds in a counter-clockwise manner until an adjacent border pixel is found. The location of the adjacent border pixel is noted and a vector is computed from the old border pixel to the new border pixel. The window is then moved to the adjacent border pixel where the search process is repeated. The search continues until the the first border pixel is encountered again. Each vector is drawn on the screen as each is found.

Encountering the first border pixel again signals a closed contour. The closed contour can either be a negative or positive area based on the resulting sum of areas under the vectors. Generally, a closed contour having with vectors tracing out a predominantly counter-clockwise path will have an area defined as negative; a predominantly clockwise path will have an area defined as positive. This area information returned to the calling routine. The centroid of the contour is also calculated.

Written by Rolando Raqueño (Center for Imaging Science)

Include Files:

Called Routines:

x_quadrant_position

```

*               y_quadrant_position               *
*               Pixel_Value_8bit                 *
*               Area_under_vector                 *
*               Draw_Vector                       *
*               Clear_Channel                     *
*
*
* Modification History:
*
* 9/16/90 Completed documentation & comments - Rolando Raqueño
* 9/30/90 Added call to Clear_Channel to clear
*         the contour trace drawn             - Rolando Raqueño
* 10/6/90 Modified to calculate centroid of
*         contour.                             - Rolando Raqueño
* 10/7/90 Modified to make Vector_Channel an
*         argument.                             - Rolando Raqueño
*****

```

IMPLICIT NONE

```

*
* PARAMETER Definitions and Declarations.
*
INTEGER*2      Image_x_Size      ! x-size of image.
INTEGER*2      Image_y_Size      ! y-size of image.
INTEGER*2      Vector_Grey_Level ! Grey level of Vector.
PARAMETER      (Image_x_Size = 512 )
PARAMETER      (Image_y_Size = 512 )
PARAMETER      (Vector_Grey_Level = 255 )

```

```

*
* Argument List Definitions
*
INTEGER*2      Vector_Channel      ! Channel to draw vector.
BYTE           Quad_tree_buffer( Image_x_Size, Image_y_Size ) ! Array.
INTEGER*2      Window_Size        ! Quad-tree resolution.
INTEGER*2      first_x_position    ! Search start x-position.
INTEGER*2      first_y_position    ! Search start y-position.
INTEGER*2      x_neutral_axis      ! x-coordinate of centroid.
INTEGER*2      y_neutral_axis      ! y-coordinate of centroid.

```

```

*
* Variable Definitions
*
INTEGER*2      Border_Value        ! Bit-plane Grey Value.
INTEGER*2      Starting_x_position ! Image x-center point.
INTEGER*2      Starting_y_position ! Image y-center point.
INTEGER*2      Center_pixel_value  ! Window center pixel.
INTEGER*2      Current_Window_x_position ! Window x-position.
INTEGER*2      Current_Window_y_position ! Window y-position.
INTEGER*2      first_border_x_position ! first border pixel x-pos.
INTEGER*2      first_border_y_position ! first border pixel y-pos.
INTEGER*2      window_x_position    ! x-position of search window.
INTEGER*2      window_y_position    ! y-position of search window.
INTEGER*2      neighbor_x_position  ! neighbor pixel x-position.
INTEGER*2      neighbor_y_position  ! neighbor pixel y-position.
INTEGER*2      next_window_x_position ! next border pixel x-position.
INTEGER*2      next_window_y_position ! next border pixel y-position.
INTEGER*2      neighbor_pixel_value ! value of neighbor pixel.
INTEGER*2      delta_x              ! delta x to neighbor pixel.
INTEGER*2      delta_y              ! delta y to neighbor pixel.
INTEGER*2      search_angle         ! search angle for border.

```

```

INTEGER*4      Number_of_border_pixels ! Number of border pixels.
INTEGER*4      Sum_of_x_coordinates    ! Running sum of x values.
INTEGER*4      Sum_of_y_coordinates    ! Running sum of y values.
LOGICAL*1      contour_area_is_open    ! Contour area is open flag.
INTEGER*2      Array_of_Points( 2,2 ) ! Start/End Coordinate Points.
REAL*4         Area                    ! Area_of_Contour

```

Function Definitions

```

INTEGER*2      x_quadrant_position      ! x-position of super-pixel.
INTEGER*2      y_quadrant_position      ! y-position of super-pixel.
INTEGER*2      Pixel_Value_8bit         ! value of pixel at (x,y).
REAL*4         Area_under_Vector        ! Calculates area under vector

```

Equivalence Definitions

```

EQUIVALENCE ( Array_of_Points(1,1), Current_Window_x_position )
EQUIVALENCE ( Array_of_Points(2,1), Current_Window_y_position )
EQUIVALENCE ( Array_of_Points(1,2), Next_Window_x_position )
EQUIVALENCE ( Array_of_Points(2,2), Next_Window_y_position )

```

```

*
* Start at the center of the image, determine quad-tree bit-plane to
* search, and start to find the first border pixel.

```

```

Starting_x_position = x_quadrant_position( first_x_position,
                                           Window_Size      ) ! start x.

```

```

Starting_y_position = y_quadrant_position( first_y_position,
                                           Window_Size      ) ! start y.

```

```

Border_Value = 128 / Window_Size          ! Calculate Border value.

```

```

Area = 0.0                                ! Initialize Contour area.
Number_of_border_pixels = 0                ! Initialize Number of pixels.
Sum_of_x_coordinates = 0
Sum_of_y_coordinates = 0

```

```

Center_pixel_value = Pixel_Value_8bit( Quad_tree_buffer,
                                         Starting_x_position,
                                         Starting_y_position ) ! Value.

```

```

! If center pixel is a border pixel,
! then return open_contour = .FALSE.

```

```

IF ( Center_Pixel_Value .GE. Border_Value ) THEN

```

```

    TYPE *, ' Center pixel is a border pixel '

```

```

    Open_Contour = Area
    x_neutral_axis = Starting_x_position
    y_neutral_axis = Starting_y_position

```

```

    RETURN

```

```

END IF

```

```

! Move left until until the first border pixel is found.

```

```

Current_Window_x_Position = Starting_x_position

```

```
Current_Window_y_Position = Starting_y_position
```

```
DO WHILE ( (Pixel_Value_8bit ( Quad_tree_buffer,  
Current_Window_x_position,  
Current_Window_y_position ))  
.LT. Border_Value )
```

```
Current_Window_x_position = Current_Window_x_position -  
Window_Size
```

```
END DO
```

```
*  
*  
*  
*
```

```
After first pixel has been found start finding the next border pixel.  
Start scanning the search window for the next border position.
```

```
First_border_x_position = Current_Window_x_position ! 1st x border.  
First_border_y_position = Current_Window_y_position ! 1st y border.  
contour_area_is_open = .TRUE. ! Assume open contour.  
Search_angle = 0 ! Starting search angle.
```

```
!Continue looking for border pixels while contour is open.  
!Contour is considered closed when the first border pixel  
!is encountered again.
```

```
DO WHILE ( contour_area_is_open )
```

```
neighbor_pixel_value = 0 ! Initialize variable.
```

```
!The direction of each neighboring pixel with respect to  
!the current border pixel is given by a search angle.
```

```
!
```

```
!The next border pixel is found by examining the neighboring  
!pixels in a counter-clockwise order every 45 degrees.
```

```
!
```

```
!Continue this search until a border pixel is found.
```

```
DO WHILE( neighbor_pixel_value .LT. border_value)
```

```
Search_angle = MOD( Search_angle + 45, 360 )
```

```
Delta_x = NINT( COSD( REAL( Search_angle ) ) ) *  
Window_Size
```

```
Delta_y = NINT( SIND( REAL( Search_angle ) ) ) *  
Window_Size
```

```
neighbor_x_position = current_window_x_position +  
delta_x
```

```
neighbor_y_position = current_window_y_position +  
delta_y
```

```
neighbor_pixel_value = Pixel_Value_8bit(  
Quad_tree_buffer,  
neighbor_x_position,  
neighbor_y_position )
```

```
END DO
```

```
!If a border pixel has been found, make a note of its  
!location, check to see if it is the same as the first  
!pixel, and move the search window to the new border pixel.
```



```

!
!With respect to the new position of the search window,
!advance the search angle for the next border pixel to start
!225 degrees counter-clockwise from the search angle made at
!the last search window position.

next_window_x_position = neighbor_x_position
next_window_y_position = neighbor_y_position

Number_of_border_pixels = Number_of_border_pixels + 1
Sum_of_x_coordinates = Sum_of_x_coordinates +
                        Next_window_x_position

Sum_of_y_coordinates = Sum_of_y_coordinates +
                        Next_window_y_position

contour_area_is_open = .NOT. ( next_window_x_position
                              .EQ.
                              first_border_x_position
                              .AND.
                              next_window_y_position
                              .EQ.
                              first_border_y_position      )

CALL Draw_Vector(      Vector_Channel,
                     Vector_Grey_Level,
                     Array_of_Points )

Area = Area + Area_under_vector ( Array_of_Points )

IF ( contour_area_is_open ) THEN

    Search_Angle = MOD( Search_Angle + 225, 360 )

    Current_Window_x_position = Next_window_x_position
    Current_Window_y_position = Next_window_y_position

END IF
END DO

x_neutral_axis =      NINT( REAL( Sum_of_x_coordinates ) /
                           REAL( Number_of_border_pixels ) )

Current_Window_x_position = x_neutral_axis
Next_Window_x_position = x_neutral_axis

y_neutral_axis =      NINT( REAL( Sum_of_y_coordinates ) /
                           REAL( Number_of_border_pixels ) )

Current_Window_y_position = y_neutral_axis
Next_Window_y_position = y_neutral_axis

CALL Draw_Vector(      Vector_Channel,
                     Vector_Grey_Level,
                     Array_of_Points )

Open_Contour = Area

RETURN

END ! Open_Contour.

```

```

SUBROUTINE      OR_8bit(      Operand1_Channel,
                             Operand2_Channel,
                             Result_Channel ) ! Logical Image OR.

```

```

*****

```

```

*
*      This subroutine performs an 8-bit image OR of two images
*      in two channels and placing the result in a third channel.
*      The subroutine utilizes the DVP to execute the bitwise
*      logical OR.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*          Attach_DVP
*          Initialize_DVP
*          DVP_OR_8bit
*          Detach_DVP
*
*      Modification History:
*
*      4/6/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INTEGER*2 Operand1_Channel      ! Channel Number of first image.
INTEGER*2 Operand2_Channel      ! Channel Number of second image.
INTEGER*2 Result_Channel        ! Channel Number of result image.

```

```

CALL Attach_DVP                  ! Allocate the DVP.
CALL Initialize_DVP              ! Initialize and Reset the DVP.

```

```

CALL DVP_OR_8bit(      Operand1_Channel,
                       Operand2_Channel,
                       Result_Channel)      ! 8-bit DVP image OR.

```

```

CALL Detach_DVP                  ! Deallocate the DVP.

```

```

RETURN

```

```

END ! OR_8bit.

```

```

/* Start of Pixel_Value_8bit function *****/
*
*      This function accepts a 512x512 8-bit Image array from a
*      FORTRAN routine and returns the unsigned 8-bit value as an
*      INTEGER*2 (short int). Since FORTRAN accesses its arrays in
*      a column order and C access its arrays in a row order, the
*      indices are swapped in order to properly access the pixel in
*      question.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*
*      Modification History:
*
*      9/10/90 Completed documentation & comments - Rolando Raqueño
*
*****/

#define MAX_ROW 512
#define MAX_COLUMN 512

unsigned      short   int   Pixel_Value_8bit(
                                unsigned char buffer_512x512x8bit[MAX_ROW][MAX_COLUMN]
                                unsigned short int *ptr_x_position,
                                unsigned short int *ptr_y_position
                                )
{
    return( buffer_512x512x8bit[*ptr_y_position][*ptr_x_position] );
}

/* End of Pixel_Value_8bit function *****/

```

```

SUBROUTINE Quadrant_Histogram_Buffer( Channel,
.                                     X_LLC,
.                                     Y_LLC,
.                                     Window_Size,
.                                     Histogram_Buffer)

```

```

*****
*
*   This subroutine is a call to generate a histogram distribution
*   for a square region of interest of a given size and location
*   in a specific channel. The result is 256 element array
*   containing the tallies. This routine is a more specific
*   version of Histogram_Channel_ROI (which is called by this
*   routine) in that it limits its regions of interest to square
*   shapes whereas Histogram_Channel_ROI can use any rectangular
*   shape.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Histogram_Channel_ROI
*
*   Modification History:
*
*   5/31/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INTEGER*2 X_LLC           ! X lower left corner point.
INTEGER*2 Y_LLC           ! Y lower left corner point.
INTEGER*2 Window_Size     ! Square window size [pixels]
INTEGER*2 Channel         ! Channel to histogram.
INTEGER*2 X_URC           ! X upper right corner point.
INTEGER*2 Y_URC           ! Y upper right corner point.

```

```

integer*4 histogram_buffer( 0:255 )    ! 256 Grey-Level Histogram.

```

```

X_URC = X_LLC + Window_Size - 1      ! X upper right corner point.
Y_URC = Y_LLC + Window_Size - 1      ! Y upper right corner point.

```

```

CALL Histogram_Channel_ROI( Histogram_Buffer,
.                             Channel,
.                             X_LLC,
.                             Y_LLC,
.                             X_URC,
.                             Y_URC )    ! Get histogram.

```

RETURN

END ! Quadrant_Histogram_Buffer


```

SUBROUTINE Quadrant_Histogram_Statistics(      Histogram_Buffer,
.                                             Total,
.                                             Low,
.                                             High,
.                                             Maximum )

```

```

*****
*
*   This function takes a 256 grey-level histogram buffer and
*   calls routines to calculate histogram statistics such as
*   total number of pixels, lowest grey-level, highest grey-
*   level, and maximum number of pixels for a given grey level.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Histogram_Total
*       Histogram_Low
*       Histogram_High
*       Histogram_Maximum
*
*   Modification History:
*
*   12/14/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

integer*4 total           ! Total pixels tallied.
integer*4 low             ! Lowest grey value tallied.
integer*4 high            ! Highest grey value tallied.
integer*4 maximum         ! Mode.

```

```

integer*4 histogram_total ! Histogram_Total Function.
integer*4 histogram_low   ! Histogram_Low Function.
integer*4 histogram_high  ! Histogram_High Function.
integer*4 histogram_maximum ! Histogram_Maximum Function

```

```

integer*4 histogram_buffer( 0:255 )    ! 256 Grey-level histogram.

```

```

total = histogram_total( histogram_buffer )
high = histogram_high( histogram_buffer )
low = histogram_low( histogram_buffer )
maximum = histogram_maximum( histogram_buffer )

```

RETURN

END ! Quadrant_Histogram_Statistics

```

/* Start of quadrant_mask function *****/
*
*   This function is simply a C to FORTRAN link to pass the
*   necessary parameters by value into a C routine which will in
*   turn relay the parameters by reference to a FORTRAN routine to
*   display the super-pixels of the appropriate size, grey=level,
*   and channel. This C to FORTRAN buffer is used to insure that
*   values during recursion are preserved in the stack and not
*   inadvertently changed by FORTRAN's argument referencing side-
*   effect.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       square_mask_11c
*
*   Modification History:
*
*   8/12/90  Completed documentation & comments - Rolando Raqueño
*
*****/

/*-----
FUNCTION PROTOTYPE SECTION-----*/
extern void square_mask_11c(    unsigned short int *ptr_grey_level,
                                unsigned short int const *ptr_channel,
                                unsigned short int *ptr_x_11c,
                                unsigned short int *ptr_y_11c,
                                unsigned short int *ptr_window_size );

/*-----
QUADRANT_MASK FUNCTION MAIN BODY-----*/
void quadrant_mask(    unsigned short int grey_level,
                        unsigned short int const channel,
                        unsigned short int x_11c,
                        unsigned short int y_11c,
                        unsigned short int window_size )
{
    square_mask_11c(    &grey_level,        /* Call to a FORTRAN */
                        &channel,            /* routine which      */
                        &x_11c,              /* makes a square mask*/
                        &y_11c,              /* in the appropriate */
                        &window_size        );/* specifications.   */
}

/* End of quadrant_mask function *****/

```

```

      INTEGER*2 FUNCTION Quadrant_Pixel_Code (      Channel,
      .                                           X_LLC,
      .                                           Y_LLC,
      .                                           Window_Size)

```

```

*****
*
*   The following function checks to see if a specified quadrant
*   contains all zero pixels ( code = 1 ), all 255 pixels
*   ( code = 2 ), or both 0 and 255 pixels ( code = 3 ).
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       quadrant_histogram_buffer
*       quadrant_histogram_statistics
*       quadrant_statistics_code
*
*   Modification History:
*
*   12/13/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

      INTEGER*2 Channel      ! Channel to check.
      INTEGER*2 X_LLC      ! x-lower left corner.
      INTEGER*2 Y_LLC      ! y-lower left corner.
      INTEGER*2 Window_Size ! Size of window to check.

      INTEGER*4 total      ! Total number of pixels.
      INTEGER*4 low        ! Lowest grey value tallied.
      INTEGER*4 high       ! Highest grey value tallied.
      INTEGER*4 maximum     ! Mode.
      INTEGER*4 Histogram_Buffer( 0:255 ) ! 256 Grey-level histogram.
      INTEGER*2 Quadrant_Statistics_Code ! Function call for code.

```

```

      CALL Quadrant_Histogram_Buffer( Channel,
      .                               X_LLC,
      .                               Y_LLC,
      .                               Window_Size,
      .                               Histogram_Buffer ) ! Get buffer.

```

```

      CALL Quadrant_Histogram_Statistics( Histogram_Buffer,
      .                               Total,
      .                               Low,
      .                               High,
      .                               Maximum ) ! Get statistics.

```

```

      Quadrant_Pixel_Code = Quadrant_Statistics_Code( Total,
      .                                               Low,
      .                                               High,
      .                                               Maximum ) ! Get Code.

```

RETURN

END ! Quadrant_Pixel_Code

```

/* Start of quadrant_pixel_mask function *****/
*
* This function displays the progress of the quad-tree creation *
* by monitoring the intermediate results in a given channel. The *
* grey level of each quadrant indicates its level in the tree. *
* For leaf node quadrants having homogeneous OFF values, *
* homogeneous ON values, and a non-homogeneous values, *
* the quadrant pixels will be displayed as super-pixels having *
* 0 grey value, 255 grey value, and 128 grey value, respectively. *
*
* Written by Rolando Raqueño (Center for Imaging Science) *
*
* Include Files: *
*
* Called Routines: *
*      quadrant_mask *
*
* Modification History: *
*
* 8/12/90 Completed documentation & comments - Rolando Raqueño *
* 9/24/90 Placed result_channel in argument list *
*      - Rolando Raqueño *
*
*****/

/*-----
FUNCTION PROTOTYPE SECTION-----*/
extern void quadrant_mask(      unsigned short int grey_value,
                                unsigned short int const channel,
                                unsigned short int x_llc,
                                unsigned short int y_llc,
                                unsigned short int window_size );

/*-----
QUADRANT_PIXEL_MASK FUNCTION MAIN BODY-----*/
void quadrant_pixel_mask(      unsigned short int result_channel,
                                unsigned short int quadrant_pixel_status_code,
                                unsigned short int x_llc,
                                unsigned short int y_llc,
                                unsigned short int window_size )
{
    unsigned      short      int      grey_value;

    switch ( quadrant_pixel_status_code )
    {
        case 1:
            grey_value = 0;
            break;          /* All quadrant values are 0's (OFF) */

        case 2:
            grey_value = 255;
            break;          /* All quadrant values are 1's (ON) */

        case 3:
            grey_value = 128 / window_size;
            break;          /* Quadrant values are 0's and 1's */

        default:
            printf( "Error in Quadrant Pixel Mask Routine");
    }
}

```



```

    }

    quadrant_mask( grey_value,          /* Display quadrant pixel */
                   result_channel,      /* value indicator in the */
                   x_llc,               /* the graphics channel   */
                   y_llc,               /* to show progress of    */
                   window_size         /* quad-tree creation.    */
                   );

}

/* End of quadrant_pixel_mask function *****/

```

```

/* Start of quadrant_pixel_status function *****/
*
* This function determines the status of a given pixel at a given *
* quad-tree level. The status is determined by taking the *
* histogram of the super-pixel ( quadrant-pixel ) and checking *
* if the pixel is homogeneously one value, i.e., ON or OFF. *
*
* A quadrant-pixel comprised of all ON values return a status *
* code of 1 indicating a virtual leaf node. *
*
* A quadrant-pixel comprised of all OFF values return a status *
* code of 2 indicating a virtual leaf node. *
*
* A quadrant-pixel having both ON and OFF values return a status *
* code of 3 indicating a mixed pixel requiring further *
* subdivision. *
*
* Written by Rolando Raqueño (Center for Imaging Science) *
*
* Include Files: *
*
* Called Routines: *
*         quadrant_histogram_buffer *
*         histogram_total *
*
* Modification History: *
*
* 8/11/90 Completed documentation & comments - Rolando Raqueño *
*
*****/

/*-----
FUNCTION PROTOTYPE SECTION-----
*/
extern void quadrant_histogram_buffer(
    unsigned short int const *ptr_channel,
    unsigned short int *ptr_x_lower_left_corner,
    unsigned short int *ptr_y_lower_left_corner,
    unsigned short int *ptr_window_size,
    unsigned int histogram_buffer[] );

extern unsigned int histogram_total( unsigned int histogram_buffer[] );

/*-----
QUADRANT_PIXEL_STATUS FUNCTION MAIN BODY-----
*/
short int quadrant_pixel_status(
    unsigned short int const channel,
    unsigned char bit_level,
    unsigned short int x_position,
    unsigned short int y_position,
    unsigned short int window_size )
{
    unsigned int histogram_buffer[256];
    /* Histogram */
    /* buffer */

    unsigned int lo_count = 0; /* Lowest Grey */
    /* Level Count */

    unsigned int hi_count = 0; /* Highest Grey */
    /* Level Count */

```

```

unsigned          int    total_count;    /* Total number */
                                           /* of pixels    */
                                           /* counted     */

unsigned          short  int    grey_level;    /* Grey level */

if ( bit_level == 0 )    /* Bit levels below 0 indicate super-pixels */
{                        /* with very low resolutions and offer no */
    return( 3 );        /* significant information. Return code to */
}                        /* signal continuation to higher resolutions*/

quadrant_histogram_buffer(    &channel,        /* Compute */
                              &x_position,    /* histogram */
                              &y_position,    /* and load */
                              &window_size,   /* buffer */
                              histogram_buffer );

total_count = histogram_total( histogram_buffer );
                                           /* Compute number of pixels counted */

/*~~~~~
The following loop goes through the histogram and sums all the
pixel counts for the grey levels that have the same bit set as
the bit_level argument. It also sums the pixels that do not
have the same bit set as the bit level argument. The purpose of
this loop is to determine the fraction of ON pixels vs. OFF
pixels for a given quadrant at a given resolution level.
*/
for ( grey_level = 0; grey_level <= 255; grey_level++ )
{
    if( grey_level & bit_level )
    {
        hi_count += histogram_buffer[grey_level];
    }
    else
    {
        lo_count += histogram_buffer[grey_level];
    }
}
/*~~~~~
The following statements returns the appropriate code for the
quadrant at the given resolution level.
*/
if( lo_count == total_count )
{
    return( 1 );    /* All pixels in the quadrant are 0's */
}

if( hi_count == total_count )
{
    return( 2 );    /* All pixels in the quadrant are 1's */
}

if( (lo_count != total_count) & (hi_count != total_count) )
{
    return( 3 );    /* Pixels in the quadrant are 1's and 0's */
}

printf( "Error determining quadrant_pixel_status \n" );
}
/* End of quadrant_pixel_status function *****/

```

```

      INTEGER*2 FUNCTION Quadrant_Statistics_Code(      Total,
      .                                              Low,
      .                                              High,
      .                                              Maximum )

```

```

*****
*
*      This function takes histogram statistics such as the total
*      number of pixels, lowest grey-level, highest grey-level,
*      and maximum number of pixels for a given grey level, and
*      determines whether or not the pixels in the region of interest
*      are all zeroes ( code = 1 ), all 255's (code = 2 ) or both
*      0's and 255's ( code = 3 ).
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*
*      Modification History:
*
*      12/14/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

integer*4 total      ! Total pixels tallied.
integer*4 low        ! Lowest grey value tallied.
integer*4 high       ! Highest grey value tallied.
integer*4 maximum    ! Mode.

```

```

IF (      (total.eq.maximum)
.          .and.
.          (high.eq.low)
.          .and.
.          (high.eq. 0 ) ) THEN

```

```

      Quadrant_Statistics_Code = 1      ! All pixels have zero value.
END IF

```

```

IF (      (total.eq.maximum)
.          .and.
.          (high.eq.low)
.          .and.
.          (high.eq. 255 ) ) THEN

```

```

      Quadrant_Statistics_Code = 2      ! All pixels have 255 value.
END IF

```

```

IF      (total.ne.maximum) THEN

```

```

      Quadrant_Statistics_Code = 3      ! Pixels are both 0 and 255.
END IF

```

RETURN

END ! Quadrant_Statistics_Code


```

SUBROUTINE Rectangular_Mask_LLC_URC(   Grey_Level,
.                                     Channel,
.                                     X_LLC,
.                                     Y_LLC,
.                                     X_URC,
.                                     Y_URC ) ! Rectangular Mask.

```

```

*****
*
*   This subroutine displays a rectangular mask having a specified
*   grey level and size in a given channel. The bounds of the
*   rectangular mask is given by a lower-left hand coordinate and
*   an upper right-hand coordinate. These Grey_Level value,
*   Channel number, and bounding coordinates are passed on to the
*   Constant_Channel_ROI routine which deposits a constant value to
*   a rectangular Region Of Interest.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Constant_Channel_ROI
*
*   Modification History:
*
*   8/13/90 Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INTEGER*2 Grey_Level      ! Grey level to set mask.
INTEGER*2 channel         ! Channel to display mask.
INTEGER*2 X_LLC           ! Lower left-hand corner x-coordinate.
INTEGER*2 Y_LLC           ! Lower left-hand corner y-coordinate.
INTEGER*2 X_URC           ! Upper right-hand corner x-coordinate.
INTEGER*2 Y_URC           ! Upper right-hand corner y-coordinate.

```

```

CALL Constant_Channel_OR_ROI(   Grey_Level,
.                               Channel,
.                               X_LLC,
.                               Y_LLC,
.                               X_URC,
.                               Y_URC   ) ! Constant Channel ROI.

```

RETURN

END ! Rectangular_Mask_LLC_URC

SUBROUTINE Region_of_Interest(X_LLC, Y_LLC, X_URC, Y_URC)

```
*****
*
*   This subroutine defines a rectangular region of interest which
*   designates the area on which subsequent DUP operations will be
*   applied. The rectangular coordinate is defined by a lower
*   left corner point and an upper right corner point.
*   This is accomplished through an IPI library call.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_DUPRegion
*       IPI_ErrorCheck
*
*   Modification History:
*
*   5/30/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.

INTEGER*4 Unit ! IP8500 Unit number.

INTEGER*2 X_LLC ! X Lower Left Corner Coordinate.

INTEGER*2 Y_LLC ! Y Lower Left Corner Coordinate.

INTEGER*2 X_URC ! X Upper Right Corner Coordinate.

INTEGER*2 Y_URC ! Y Upper Right Corner Coordinate.

COMMON / Ipi_Unit / Unit ! IPI_Unit common block.

Status = Ipi_DUPRegion(Unit, X_LLC, Y_LLC, X_URC, Y_URC)

CALL Ipi_ErrorCheck(Status, 'Error Setting Region of Interest')

RETURN

END ! Region_of_Interest

SUBROUTINE Reset_View_Channel

```
*****
*
*   This subroutine resets viewing of channels to normal RGB.
*
*   Image Processing Interface Library (IPI) call
*       developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_ViewChan
*       IPI_ErrorCheck
*
*   Modification History:
*
*   9/27/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INCLUDE 'IPI_IPILIB'      ! Include IPI Header File.
INTEGER*4 Status          ! Status call variable.
INTEGER*4 Unit            ! IP8500 Unit Number.
```

```
COMMON /Ipi_Unit/ Unit    ! IPI_Unit common block.
```

```
Status = Ipi_ViewChan( Unit )
CALL Ipi_ErrorCheck(Status,'Error Resetting View Channel')
```

RETURN

END ! Reset_View_Channel

```

SUBROUTINE      Save_Picture_8bit(      File_Name,
                                          Channel_Number )

```

```

*****
*
*      This subroutine saves an image in a memory channel based
*      on the Picture options set in the PicOps variable.
*
*      Image Processing Interface Library (IPI) call
*      developed by S. Schultz
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*
*      Include Files:
*
*          IPI_IPILIB
*
*
*      Called Routines:
*
*          IPI_SavePic
*          IPI_ErrorCheck
*
*
*      Modification History:
*
*      4/7/90  Completed documentation & comments - Rolando Raqueño
*      7/23/90 Added IPI_PicOps common block - Rolando Raqueño
*      8/2/90  Renamed to Save_Picture_8bit - Rolando Raqueño
*      9/30/90 Changed channel_mask argument to a channel_number
*               argument                      - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INCLUDE 'IPI_IPILIB'      ! IPI header file.

INTEGER*4 Status           ! Status call variable.
INTEGER*4 Unit             ! IP8500 unit number.
INTEGER*4 PicOps           ! Picture Option settings.
CHARACTER*80 File_Name     ! Image file name.
INTEGER*2 Channel_Number   ! Channel number to save.
INTEGER*4 Channel_Mask     ! Bitwise Channel selection.

COMMON /IPI_Unit/ Unit      ! IPI_Unit common block.
COMMON /IPI_PicOps/ PicOps ! IPI_PicOps common block.

Channel_Mask = 2 ** Channel_Number

Status = Ipi_SavePic( Unit, File_Name, Channel_Mask, PicOps )
CALL Ipi_ErrorCheck( Status, 'Error Saving Picture' )

RETURN

END ! Save_Picture_8bit

```



```

SUBROUTINE      Save_Picture_16bit(      File_Name, Channel_Mask )

*****
*
*      This subroutine saves a 16-bit image from two of the IP8500's
*      memory channels based on the Picture Options previously set.
*
*      Image Processing Interface Library (IPI) call
*      developed by S. Schultz
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*              IPI_IPILIB
*
*      Called Routines:
*
*              IPI_SavePic
*              IPI_ErrorCheck
*
*      Modification History:
*
*      7/24/90  Completed documentation & comments - Rolando Raqueño
*
*****

IMPLICIT NONE

INCLUDE 'IPI_IPILIB'           ! IPI header file.

INTEGER*4 Status                ! Status call variable.
INTEGER*4 Unit                  ! IP8500 unit number.
INTEGER*4 PicOps                ! Picture Option settings.
CHARACTER*80 File_Name          ! Image file name.
INTEGER*4 Channel_Mask(2)       ! Bitwise Channel selection.

COMMON /IPI_Unit/ Unit          ! IPI_Unit common block.
COMMON /IPI_PicOps/ PicOps      ! IPI_PicOps common block.

Status = Ipi_SavePic( Unit, File_Name, Channel_Mask, PicOps )
CALL Ipi_ErrorCheck( Status, 'Error Saving Picture' )

Return

END ! Save_Picture_16bit

```

SUBROUTINE Scroll_Channel(X_Value, Y_Value, Channel_Mask)

```
*****
*
*   This subroutine scrolls the bitwise selected channels as
*   defined by the Channel_Mask. The scroll is accomplished by
*   moving the pixel at location ( X_Value, Y_Value ) to the
*   upper left hand corner of the display screen. This is
*   accomplished by an
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_Scroll
*
*   Modification History:
*
*   4/6/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INCLUDE 'IPI_IPILIB'           ! Include IPI Header File.

INTEGER*4 Status                ! Status call variable.
INTEGER*4 Unit                  ! IP8500 Unit number.
INTEGER*4 Channel_Mask          ! Bitwise channel selection variable.
INTEGER*2 X_Value               ! X Scroll Register Variable.
INTEGER*2 Y_Value               ! Y Scroll Register Variable.

COMMON /IPI_Unit/ Unit          ! IPI_Unit common block.

Status = Ipi_Scroll( Unit, Channel_Mask, X_Value, Y_Value ) ! Scroll.

CALL Ipi_ErrorCheck( Status, 'Error Scrolling Channels' )

RETURN

END ! Scroll_Channel
```

SUBROUTINE Scroll_Operand1_East_16bit ! Scroll Right 1-pixel.

```
*****
*
*   This subroutine scrolls the 16-bit operand in channels (0,1)
*   one pixel to the right.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Scroll_Channels
*
*   Modification History:
*
*   7/27/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel_Mask      ! Bitwise Channel selection
INTEGER*2 X_Scroll          ! X_Scroll Register.
INTEGER*2 Y_Scroll          ! Y_Scroll Register.
```

```
PARAMETER (X_Scroll = -1)    ! Shift Image right 1 pixel
PARAMETER (Y_Scroll = 511)   ! No shift up and down.
PARAMETER (Channel_Mask = 3) ! Scroll channels (0,1).
```

```
CALL Scroll_Channels( X_Scroll,
                     Y_Scroll,
                     Channel_Mask ) ! Scroll Right 1 pixel.
```

RETURN

END ! Scroll_Operand1_East_16bit

SUBROUTINE Scroll_Operand1_West_16bit ! Scroll Left 1-pixel.

```
*****
*
*   This subroutine scrolls the 16-bit operand in channels (0,1)
*   one pixel to the left.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Scroll_Channels
*
*   Modification History:
*
*   7/27/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel_Mask           ! Bitwise Channel selection
INTEGER*2 X_Scroll               ! X_Scroll Register.
INTEGER*2 Y_Scroll               ! Y_Scroll Register.
```

```
PARAMETER (X_Scroll = 1)        ! Shift Image left 1 pixel.
PARAMETER (Y_Scroll = 511)      ! No shift up and down.
PARAMETER (Channel_Mask = 3)    ! Scroll channels (0,1).
```

```
CALL Scroll_Channels( X_Scroll,
                     Y_Scroll,
                     Channel_Mask ) ! Scroll Left 1 pixel.
```

RETURN

END ! Scroll_Operand1_West_16bit

SUBROUTINE Scroll_Operand1_North_16bit ! Scroll Up 1-pixel.

```
*****
*
*   This subroutine scrolls the 16-bit operand in channels (0,1)
*   one pixel up.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Scroll_Channels
*
*   Modification History:
*
*   7/27/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel_Mask      ! Bitwise Channel selection
INTEGER*2 X_Scroll          ! X_Scroll Register.
INTEGER*2 Y_Scroll          ! Y_Scroll Register.
```

```
PARAMETER (X_Scroll = 0)    ! No shift left and right.
PARAMETER (Y_Scroll = 510)  ! Shift image up one pixel.
PARAMETER (Channel_Mask = 3) ! Scroll channels (0,1).
```

```
CALL Scroll_Channels( X_Scroll,
                     Y_Scroll,
                     Channel_Mask ) ! Scroll Up 1 pixel.
```

RETURN

END ! Scroll_Operand1_North_16bit

SUBROUTINE Scroll_Operand1_South_16bit ! Scroll Down 1-pixel.

```

*****
*
*   This subroutine scrolls the 16-bit operand in channels (0,1)
*   one pixel to the right.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Scroll_Channels
*
*   Modification History:
*
*   7/27/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INTEGER*4 Channel_Mask      ! Bitwise Channel selection.
INTEGER*2 X_Scroll          ! X_Scroll Register.
INTEGER*2 Y_Scroll          ! Y_Scroll Register.

```

```

PARAMETER (X_Scroll = 0)    ! No shift left or right.
PARAMETER (Y_Scroll = 512)  ! Shift down 1 pixel.
PARAMETER (Channel_Mask = 3) ! Scroll channels (0,1).

```

```

CALL Scroll_Channels( X_Scroll,
                     Y_Scroll,
                     Channel_Mask ) ! Scroll Down 1 pixel.

```

RETURN

END ! Scroll_Operand1_South_16bit

SUBROUTINE Scroll_Operand1_SouthEast_16bit ! Scroll Down-Right.

```
*****
*
*      This subroutine scrolls the 16-bit operand in channels (0,1)
*      one pixel down and to the right.
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*          Scroll_Channels
*
*      Modification History:
*
*      7/27/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel_Mask      ! Bitwise Channel selection.
INTEGER*2 X_Scroll          ! X_Scroll Register.
INTEGER*2 Y_Scroll          ! Y_Scroll Register.
```

```
PARAMETER (X_Scroll = -1)   ! Shift Image right 1 pixel.
PARAMETER (Y_Scroll = 512)  ! Shift Image down 1 pixel.
PARAMETER (Channel_Mask = 3) ! Scroll channels (0,1).
```

```
CALL Scroll_Channels( X_Scroll,
                     Y_Scroll,
                     Channel_Mask ) ! Scroll Right-Down 1 pixel.
```

RETURN

END ! Scroll_Operand1_SouthEast_16bit

SUBROUTINE Scroll_Operand1_SouthWest_16bit ! Scroll Down-Left.

```
*****
*
* This subroutine scrolls the 16-bit operand in channels (0,1)
* one pixel down and to the left.
*
* Written by Rolando Raqueño (Center for Imaging Science)
*
* Include Files:
*
* Called Routines:
*             Scroll_Channels
*
* Modification History:
*
* 7/27/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel_Mask           ! Bitwise Channel selection.
INTEGER*2 X_Scroll               ! X_Scroll Register.
INTEGER*2 Y_Scroll               ! Y_Scroll Register.
```

```
PARAMETER (X_Scroll = 1)         ! Shift Image left 1 pixel.
PARAMETER (Y_Scroll = 512)       ! Shift Image down 1 pixel.
PARAMETER (Channel_Mask = 3)     ! Scroll channels (0,1).
```

```
CALL Scroll_Channels( X_Scroll,
                     Y_Scroll,
                     Channel_Mask ) ! Scroll Down-Left 1 pixel.
```

RETURN

END ! Scroll_Operand1_SouthWest_16bit

SUBROUTINE Scroll_Operand1_NorthWest_16bit ! Scroll Up-Left.

```
*****
*
*   This subroutine scrolls the 16-bit operand in channels (0,1)
*   one pixel up and to the right.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Scroll_Channels
*
*   Modification History:
*
*   7/27/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel_Mask      ! Bitwise Channel selection
INTEGER*2 X_Scroll          ! X_Scroll Register.
INTEGER*2 Y_Scroll          ! Y_Scroll Register.
```

```
PARAMETER (X_Scroll = 1)    ! Shift Image left 1 pixel.
PARAMETER (Y_Scroll = 510)  ! Shift Image up 1 pixel.
PARAMETER (Channel_Mask = 3) ! Scroll channels (0,1).
```

```
CALL Scroll_Channels( X_Scroll,
                     Y_Scroll,
                     Channel_Mask ) ! Scroll Up-Left 1 pixel.
```

RETURN

END ! Scroll_Operand1_NorthWest_16bit

SUBROUTINE Scroll_Operand1_NorthEast_16bit ! Scroll Up-Right.

```
*****
*
*   This subroutine scrolls the 16-bit operand in channels (0,1)
*   one pixel up and to the right.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Scroll_Channels
*
*   Modification History:
*
*   7/27/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Channel_Mask      ! Bitwise Channel selection.
INTEGER*2 X_Scroll          ! X_Scroll Register.
INTEGER*2 Y_Scroll          ! Y_Scroll Register.
```

```
PARAMETER (X_Scroll = -1)   ! Shift Image right 1 pixel.
PARAMETER (Y_Scroll = 510)  ! Shift Image up 1 pixel.
PARAMETER (Channel_Mask = 3) ! Scroll channels (0,1).
```

```
CALL Scroll_Channels( X_Scroll,
.                      Y_Scroll,
.                      Channel_Mask ) ! Scroll Up-Right 1 pixel.
```

RETURN

END ! Scroll_Operand1_NorthEast_16bit

SUBROUTINE Set_Mono_Picture_Options ! Monochrome Defaults

```
*****
*
*   This subroutine sets the IP8500 system picture display to
*   Standard monochrome image display options by an IPI Library
*   call. Standard monochrome image is currently defined as a
*   512 x 512 x 8 bit image.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_SetPicOps
*       IPI_ErrorCheck
*
*   Modification History:
*
*   5/31/90  Completed documentation & comments - Rolando Raqueño
*   7/22/90  Added IPI_PicOps common block - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.
INTEGER*4 PicOps ! Picture Options variable.

COMMON /IPI_PicOps/ PicOps ! IPI_PicOps common block.

Status = Ipi_SetPicOps(PicOps) ! Set Picture Options call.

CALL Ipi_ErrorCheck(Status, 'Error Setting Picture Options')

RETURN

END ! Set_Mono_Picture_Options

SUBROUTINE Set_Word_Picture_Options ! 512x512x16 bit image.

```
*****
*
* This subroutine sets the IP8500 system picture display to
* 16-bit word display options by an IPI Library call.
* The dimension of the image 512 x 512 x 16 bit image.
*
* Image Processing Interface Library (IPI) call
* developed by S. Schultz
*
* Written by Rolando Raqueño (Center for Imaging Science)
*
* Include Files:
* IPI_IPILIB
*
* Called Routines:
* IPI_SetPicOps
* IPI_ErrorCheck
*
* Modification History:
*
* 7/24/90 Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.

INTEGER*4 PicOps ! Picture Options variable.

COMMON /IPI_PicOps/ PicOps ! IPI_PicOps common block.

Status = Ipi_SetPicOps(PicOps, IPI_M_WordData)

CALL Ipi_ErrorCheck(Status, 'Error Setting Picture Options')

RETURN

END ! Set_Word_Picture_Options

SUBROUTINE set_y_axis_inverted ! Set Y-axis inverted.

```
*****
*
* This subroutine sets the IP8500 system picture display to
* invert the y-axis when the image is accessed. Also, this
* routine can be used to align the screen coordinates of the
* IP8500 screen channels with the array coordinates of program
* memory.
*
*          (511,511)          (0,0)
*          +-----+          +-----+
*          |         |          |         |
*          | IP8500   |          | Memory  |
*          | Screen   |          | Array   |
*          | Coordinates |          | Coordinates |
*          |         |          |         |
*          +-----+          +-----+
*          (0,0)  --x-->          --x--> (511,511)
*
* This can be accomplished by displaying the image normally
* using default Picture Options and then calling this routine
* to cause the loading of the image from screen memory to data
* buffer ( utilizing some form of Ipi_GetPic) load the pixels
* in coincident (x,y) coordinates. This implies that the same
* pixel value will be accessed in both coordinate spaces for
* each (x,y) location.
*
* Image Processing Interface Library (IPI) call
* developed by S. Schultz
*
* Written by Rolando Raqueño (Center for Imaging Science)
*
* Include Files:
*
*          IPI_IPILIB
*
* Called Routines:
*
*          IPI_SetPicOps
*          IPI_ErrorCheck
*
* Modification History:
*
* 9/10/90 Completed documentation & comments - Rolando Raqueño
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! Include IPI header file.

INTEGER*4 Status ! Status call variable.
 INTEGER*4 PicOps ! Picture Options variable.

COMMON /IPI_PicOps/ PicOps ! IPI_PicOps common block.

Status = Ipi_SetPicOps(PicOps,%val(PicOps), IPI_M_YInvert)
 CALL Ipi_ErrorCheck(Status, 'Error Setting Picture Options')

RETURN

END ! set_y_axis_inverted

SUBROUTINE Setup_MinMax_DVP_Inputs_16bit

```
*****
*
*   This subroutine sets up the input busses of the DVP to attach
*   the proper channels for a 16 bit pixel-by-pixel DVP Minimum or
*   Maximum operation.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_DVPInput
*
*   Modification History:
*
*   7/20/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

```
IMPLICIT NONE
INCLUDE 'IPI_IPILIB'           ! Include IPI header file.

INTEGER*4 Status               ! Status call variable.
INTEGER*4 Unit                 ! IP8500 unit number.
INTEGER*2 Operand1_L0_Channel
INTEGER*2 Operand1_H1_Channel
INTEGER*2 Operand2_L0_Channel
INTEGER*2 Operand2_H1_Channel
PARAMETER (Operand1_H1_Channel = 0)
PARAMETER (Operand1_L0_Channel = 1)
PARAMETER (Operand2_H1_Channel = 2)
PARAMETER (Operand2_L0_Channel = 3)

Common /IPI_Unit/ Unit         ! IPI_Unit common block.

Status = Ipi_DVPInput ( Unit,
.                               Operand1_L0_Channel,
.                               Operand1_H1_Channel,
.                               ,
.                               ,
.                               Operand2_L0_Channel,
.                               Operand2_H1_Channel,
.                               ,
.                               )
CALL Ipi_ErrorCheck( Status, 'Error Assigning DVP Inputs' )

RETURN

END ! Setup_MinMax_DVP_Inputs_16bit
```

SUBROUTINE Setup_MinMax_DVP_Outputs_16bit

```

*****
*
*   This subroutine sets up the output busses of the DVP to attach
*   the proper channels for a 16 bit pixel-by-pixel DVP Minimum or
*   Maximum operation.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Assign_DVP_Output_Channel
*
*   Modification History:
*
*   7/20/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INTEGER*2    Result_HI_Channel
INTEGER*2    Result_LO_Channel
INTEGER*2    DUPA
INTEGER*2    DUPB

```

```

PARAMETER ( Result_HI_Channel = 2 )    ! Channel of HI byte.
PARAMETER ( Result_LO_Channel = 3 )    ! Channel of LO byte
PARAMETER ( DUPA = 0 )                 ! Code for output DUPA.
PARAMETER ( DUPB = 1 )                 ! Code for output DUPB.

```

```

CALL Assign_DVP_Output_Channel( DUPA, Result_LO_Channel )
CALL Assign_DVP_Output_Channel( DUPB, Result_HI_Channel )

```

RETURN

END ! Setup_MinMax_DVP_Outputs_16bit

```

SUBROUTINE      Show_Picture_8bit(      File_Name,
                                         Channel_Number )

```

```

*****
*
*      This subroutine shows an image by loading it to the memory
*      channels based on the Picture Options previously set.
*
*      Image Processing Interface Library (IPI) call
*      developed by S. Schultz
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*          IPI_IPILIB
*
*      Called Routines:
*
*          IPI_ShowPic
*          IPI_ErrorCheck
*
*      Modification History:
*
*      4/7/90  Completed documentation & comments - Rolando Raqueño
*      7/22/90 Added IPI_PicOps common block - Rolando Raqueño
*      8/2/90  Changed name to Show_Picture_8bit - Rolando Raqueño
*      9/29/90 Changed Channel_Mask argument to channel_number
*              -Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INCLUDE 'IPI_IPILIB'          ! IPI header file.

INTEGER*4 Status              ! Status call variable.
INTEGER*4 Unit                ! IP8500 unit number.
INTEGER*4 PicOps              ! Picture Option settings.
CHARACTER*80 File_Name        ! Image file name.
INTEGER*2 Channel_Number      ! Channel to display image.
INTEGER*4 Channel_Mask        ! Bitwise Channel selection.

```

```

COMMON /IPI_Unit/ Unit        ! IPI_Unit common block.
COMMON /IPI_PicOps/ PicOps    ! IPI_PicOps common block.

```

```

Channel_Mask = 2 ** Channel_Number

```

```

Status = Ipi_ShowPic( Unit, File_Name, Channel_Mask, PicOps )
CALL Ipi_ErrorCheck( Status, 'Error Showing Picture' )

```

```

RETURN

```

```

END ! Show_Picture_8bit

```


SUBROUTINE Show_Picture_16bit(File_Name, Channel_Mask)

```
*****
*
*   This subroutine shows a 16-bit image by loading it to the
*   memory channels based on the Picture Options previously set.
*
*   Image Processing Interface Library (IPI) call
*   developed by S. Schultz
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_ShowPic
*       IPI_ErrorCheck
*
*   Modification History:
*
*   7/24/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

INCLUDE 'IPI_IPILIB' ! IPI header file.

INTEGER*4 Status ! Status call variable.
INTEGER*4 Unit ! IP8500 unit number.
INTEGER*4 PicOps ! Picture Option settings.
CHARACTER*80 File_Name ! Image file name.
INTEGER*4 Channel_Mask(2) ! Bitwise Channel selection.

COMMON /IPI_Unit/ Unit ! IPI_Unit common block.
COMMON /IPI_PicOps/ PicOps ! IPI_PicOps common block.

Status = Ipi_ShowPic(Unit, File_Name, Channel_Mask, PicOps)
CALL Ipi_ErrorCheck(Status, 'Error Showing Picture')

RETURN

END ! Show_Picture_16bit

```

SUBROUTINE Square_Mask_LLC(   Grey_level,
.                               Channel,
.                               X_LLC,
.                               Y_LLC,
.                               Square_Size ) ! Display Square Mask.

```

```

*****
*
*   This subroutine displays a square mask of a specified grey
*   level and size in a given channel and location within the
*   channel. The lower left-hand coordinates and the size of
*   one side of the square are used to calculate the upper right-
*   hand coordinates which are necessary arguments for the
*   subsequent call to Rectangular_Mask_LLC_URC which is a general
*   display routine of rectangular masks.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Rectangular_Mask_LLC_URC
*
*   Modification History:
*
*   8/12/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

INTEGER*2 Grey_level      ! Grey level to set square mask.
INTEGER*2 Channel         ! Channel to display mask.
INTEGER*2 X_LLC           ! Lower left-hand corner x-coordinate.
INTEGER*2 Y_LLC           ! Lower left-hand corner y-coordinate.
INTEGER*2 X_URC           ! Upper right-hand corner x-coordinate.
INTEGER*2 Y_URC           ! Upper right-hand corner y-coordinate.
INTEGER*2 Square_Size     ! Size of square mask to display.

```

```

X_URC = X_LLC + Square_Size - 1 ! Calculate LLC x-coordinate.
Y_URC = Y_LLC + Square_Size - 1 ! Calculate LLC y-coordinate.

```

```

call rectangular_mask_llc_urc( Grey_level,
.                               Channel,
.                               X_LLC,
.                               Y_LLC,
.                               X_URC,
.                               Y_URC ) ! Display square mask.

```

RETURN

END ! Square_Mask_LLC

SUBROUTINE Store_Result_16Bit(File_Name)

```

*****
*
*   This subroutine stores a 16-bit image as a 16-bit result
*   from channel (2,3).
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Clear_Scroll
*       Set_Word_Picture_Options
*       Save_Picture_16bit
*
*   Modification History:
*
*   7/25/90 Completed documentation & comments - Rolando Raqueño
*   8/6/90 Swapped Channel Mask values to conform with DEC
*         H1 and L0 order byte convention - Rolando Raqueño
*
*****

```

IMPLICIT NONE

```

CHARACTER*80 File_Name           ! 16-bit Image file name.
INTEGER*4 Channel_Mask(2:3)      ! Bitwise channel selection.

DATA Channel_Mask /8,4/          ! Select channels (2,3)

CALL Clear_Scroll                 ! Clear Scroll (All channels)
CALL Set_Word_Picture_Options     ! Specify image as 16-bits.
CALL Save_Picture_16bit( File_Name, Channel_Mask ) ! Store Image

RETURN

END ! Store_Result_16bit

```

SUBROUTINE Subtract_16bit ! 16-bit subtraction.

```
*****
*
*   This subroutine calculates a 16-bit pixel-by-pixel subtraction
*   using the DVP. The minuend image will be in channel (0,1) and
*   the subtrahend image will be in channel (2,3). The resulting
*   difference image will be deposited in channel (2,3) overwriting
*   the subtrahend image.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Attach_DVP
*       Initialize_DVP
*       DVP_Subtract_16bit
*       Detach_DVP
*
*   Modification History:
*
*   7/22/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
CALL Attach_DVP           ! Get the DVP for use.
CALL Initialize_DVP       ! Initialize the DVP.
CALL DVP_Subtract_16bit   ! Start 16-bit DVP Operation.
CALL Detach_DVP           ! Release DVP resource.
```

RETURN

END ! Subtract_16bit


```

/* Start of Sum_Image_512x512x16bit function *****/
*
*   This function accepts a 512x512 16-bit Image array and returns
*   the sum of all the pixel values as a double value. The 16-bit
*   pixel values are assumed to be unsigned.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*
*   Modification History:
*
*   8/8/90  Completed documentation & comments - Rolando Raqueño
*
*****/
#define MAX_ROW 512
#define MAX_COLUMN 512

double sum_image_512x512x16bit( unsigned short int
                                buffer_512x512x16bit [MAX_ROW]
                                                                [MAX_COLUMN] )

{
    unsigned short int row; /* Row Counter */
    unsigned short int column; /* Column Counter */
    double sum=0; /* Summing Variable */

    for ( row = 0; row < MAX_ROW; row++ )
    {
        for ( column = 0; column < MAX_COLUMN; column++ )
        {
            sum += buffer_512x512x16bit[row][column];
        }
    }
    return( sum );
}

/* End of Sum_Image_512x512x16bit function *****/

```

REAL*4 FUNCTION Sum_of_Grad_Values

! Summation.

```
*****
*
*   This function loads grad.image image file into channel 0 and
*   sums all of its grey values and returns that value as a REAL*4.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*       Load_Grad_Image
*       Histogram_Channel
*
*   Modification History:
*
*   8/5/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INTEGER*4 Histogram_Buffer(0:255)      ! 256-grey level buffer.
INTEGER*4 Channel                       ! Channel to Histogram.
REAL*4 Grey_Level_Sum                  ! Summation Variable.
INTEGER*4 Grey_Level                   ! Grey Level Value.
```

```
PARAMETER( Channel = 1 )                ! Histogram channel 1.
```

```
CALL Load_Grad_Image                    ! Load Image file.
CALL Histogram_Channel( Histogram_Buffer,
                        Channel )        ! Histogram Image.
```

```
Grey_Level_Sum = 0                      ! Zero Summation Variable.
```

```
DO Grey_Level = 0,255                  ! Loop through histogram
                                        ! and sum all values.
```

```
    Grey_Level_Sum = Grey_Level_Sum +
    (Grey_Level*Histogram_Buffer(Grey_Level))
```

```
END DO
```

```
Sum_of_Grad_Values = Grey_Level_Sum    ! Return Summation Result.
```

```
RETURN
```

```
END ! Sum_of_Grad_Values
```

```
REAL*8 FUNCTION Sum_of_Grey_Grad_Values ! Summation.
```

```
*****
*
* This function loads a 16-bit Grey_Grad Image into channel (2,3) *
* and sums all of its Grey_Grad values and returns the value as a *
* REAL*4. *
*
* Written by Rolando Raqueño (Center for Imaging Science) *
*
* Include Files: *
*
* Called Routines: *
*      Load_Grey_Grad_Image *
*      Move_Operand2_16bit_to_Buffer *
*      Sum_Image_512x512x16bit *
*
* Modification History: *
*
* 8/5/90 Completed documentation & comments - Rolando Raqueño *
* 8/8/90 Added C function Sum_Image_512x512x16bit function to *
*        utilize unsigned arithmetic capability not found in *
*        FORTRAN - Rolando Raqueño *
*
*****
```

```
IMPLICIT NONE
```

```
CHARACTER*80 File_Name ! 8-bit Image file name.
INTEGER*2 Grey_Grad_Values(0:511,0:511) ! 16-bit Image Buffer.
INTEGER*2 Row ! Row Variable.
INTEGER*2 Column ! Column Variable.

REAL*8 Sum_Image_512x512x16bit ! Function call.

CALL Load_Grey_Grad_Image ! Load Image to channel (2,3).

CALL Move_Operand2_16bit_to_Buffer( Grey_Grad_Values )
! Move channel image to buffer

Sum_of_Grey_Grad_Values = Sum_Image_512x512x16bit( Grey_Grad_Values )

RETURN

END ! Sum_of_Grey_Grad_Values
```

SUBROUTINE View_Overlay_Graphics_Channel

```
*****
*
*   This subroutine views the overlay graphics channel.
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*       IPI_IPILIB
*
*   Called Routines:
*       IPI_ViewChan
*       IPI_ErrorCheck
*
*   Modification History:
*
*   9/27/90  Completed documentation & comments - Rolando Raqueño
*
*****
```

IMPLICIT NONE

```
INCLUDE 'IPI_IPILIB'           ! Include IPI Header File.
INTEGER*4 Status                ! Status call variable.
INTEGER*4 Unit                  ! IP8500 Unit Number.
```

```
INTEGER*2 Overlay_Graphics_Channel ! Graphics Channel.
```

```
PARAMETER ( Overlay_Graphics_Channel = 3 ) ! Channel 3.
```

```
COMMON /Ipi_Unit/ Unit          ! IPI_Unit common block.
```

```
Status = Ipi_ViewChan( Unit, Overlay_Graphics_Channel )
CALL Ipi_ErrorCheck( Status, 'Error Viewing Overlay Graphics Channel' )
```

RETURN

END ! View_Overlay_Graphics


```

      INTEGER*2      FUNCTION x_Quadrant_Position (  x_position,
                                                    window_size )

*****
*
*      This function returns the full resolution x-coordinate of the
*      center point of the quadrant pixel ( i.e. super-pixel ) having
*      window_size by window_size dimensions [full resolution pixels]
*
*      Written by Rolando Raqueño (Center for Imaging Science)
*
*      Include Files:
*
*      Called Routines:
*
*      Modification History:
*
*      9/10/90  Completed documentation & comments - Rolando Raqueño
*
*****

      IMPLICIT NONE

      INTEGER*2      x_position      ! Full resolution x-position.
      INTEGER*2      window_size     ! Dimension of quadrant pixel.

      x_quadrant_position =  (x_position / window_size) * window_size
                           + ( window_size / 2 )  ! Center coordinate

      RETURN

      END ! x_Quadrant_Position

```

```

INTEGER*2      FUNCTION y_Quadrant_Position (  y_position,
                                              window_size )

```

```

*****
*
*   This function returns the full resolution y-coordinate of the
*   center point of the quadrant pixel ( i.e. super-pixel ) having
*   window_size by window_size dimensions [full resolution pixels]
*
*   Written by Rolando Raqueño (Center for Imaging Science)
*
*   Include Files:
*
*   Called Routines:
*
*   Modification History:
*
*   9/10/90  Completed documentation & comments - Rolando Raqueño
*
*****

```

```

IMPLICIT NONE

```

```

INTEGER*2      y_position      ! Full resolution y-position.
INTEGER*2      window_size     ! Dimension of quadrant pixel.

```

```

y_quadrant_position = (y_position / window_size) * window_size
                     + ( window_size / 2 ) ! Center coordinate.

```

```

RETURN

```

```

END ! y_Quadrant_Position

```

Appendix D:
Image Processing Interface Library
Help File

Ipi_AttUnit - Attach Unit

This routine attaches the Unit and returns the Channel into the Unit argument. The standard Unit is "IPO" although the user may optionally supply an alternate name.

Format: Ipi_AttUnit(Unit [, AltName])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Arguments: Unit
Usage: Unit to Assign and Attach
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Reference
Specifies the Unit to attach the DVP to. Unit is returned by Ipi_AttUnit.

AltName
Usage: Alternate Unit Name
Type: Character Buffer
Access: Read Only
Mechanism: By Descriptor
If AltName is not present, then Ipi_AttUnit looks for "IPO", if AltName is present, then Ipi_AttUnit looks for a device with that name.

Description:

Ipi_AttUnit assigns an I/O Channel to the Device indicated by either the default or the Alternate Name and then attaches it to the process. It returns the I/O Channel number through the Unit argument so that it may be passed to any of the other IPI routines that require use of the Device.

Condition Values Returned:

SS\$_Normal Normal successful completion
Ipi_BadParCnt Bad parameter count
Any of those returned by Ipi_Qiow or Sys\$Assign

Modification History:

Date	Auth	Ver	Modification
21-Oct-1986	SLS	1.0	All new Code
27-Jan-1987	SLS	1.1	Changed over to New IpiInterface

Authors: Stephen L. Schultz

Example:

```
Program ClearChanOne
*
***** Define Unit, and include the Ipi Definitions
*
Integer*4 Unit
Include 'Ipi_IpiLib'
*
***** Attach the Unit
*
Call Ipi_AttUnit( Unit )
*
```



```
*****      Use Ipi_Scroll to View Sector 2 of Channel One
*
      Call Ipi_Scroll( Unit, 0, 1023 )
*
*****      Detatch the Unit
*
      Call Ipi_DetUnit( Unit )
      End
```

Ipi_AttDvp - Attatch Dvp

This routine attatches the DVP to the Unit the user has specified.

Format: Ipi_AttDvp(Unit [, Wait])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Arguments: Unit
Usage: Unit to Attatch Dvp to
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the Unit to attach the DVP to. Unit is returned by Ipi_AttUnit.

Wait
Usage: Indicate the preference to wait
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Wait is a flag used to indicate the action to take when the DVP is already busy. If the Wait flag is not present, then the routine will return immediately with the SS\$_DevActive return status. If the Wait flag is present, the routine will wait for the DVP to free up and then continue. A message will be printed to the screen if the Wait flag is a one, or the message is suppressed if the Wait flag is a zero.

Description:

Ipi_AttDvp attatches the DVP, or Digital Video Processor, to the Unit supplied. The DVP is a critical resource and must only be used by one Unit at a time. Because of this, it is recommended that the user Attatch it just prior to the section of his routine that uses the DVP and then Detaches it immediately following the section. This will help prevent one user locking the DVP for long periods of time even though he is not utilizing it. Ipi_DetDvp is used to detach the DVP. If the Wait flag is present and the device is already active, when it frees up, the routine will return SS\$_Normal, not SS\$_DevActive since the routine did complete normally, it just took a little longer. The values for SS\$_DevActive are as follows:

SS\$_DevActive, 000708 decimal, 0002C4 hexadecimal, 001304 octal
%System-F-DevActive, device is active

Condition Values Returned:

SS\$_Normal	Normal successful completion
SS\$_DevActive	Device is active
Ipi_BadParCnt	Bad parameter count
Any of those returned by Ipi_Qiow or Sys\$SeTimr	

Modification History:

Date	Auth	Ver	Modification
21-Oct-1986	SLS	1.0	All new Code
27-Jan-1987	SLS	1.1	Changed over to New IpiInterface

Authors: Stephen L. Schultz

Example:

```
Program ClearChanOne
*
*****      Define Unit, and include the Ipi Definitions
*
Integer*4 Unit
Include 'Ipi_IpiLib'
*
*****      Attatch the Unit and the DUP
*
Call Ipi_AttUnit( Unit )
Call Ipi_AttDvp( Unit, 1 )
*
*****      Use DvpMath to Clear Channel One
*
Call Ipi_DvpMath( Unit, Ipi_Clear, 1, 1, 1 )
*
*****      Detatch the DUP and the Unit
*
Call Ipi_DetDvp( Unit )
Call Ipi_DetUnit( Unit )

End
```

Ipi_CalcHst - Calculate a Histogram

This routine calculates a Histogram for a given channel. Note that this requires use of the DVP and the User must therefore have made a call to Ipi_AttDvp prior to making this call.

Format: Ipi_CalcHst(Unit, Chan [, HstOps])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Arguments: Unit
Usage: Unit to Perform the Operation On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the Unit to Calculate a Histogram on.

Chan
Usage: Channel to Calculate a Histogram of.
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the Channel to Calculate the Histogram of. A value of 0 indicates channel zero, a value of 1 indicates channel one, etcetera.

HstOps
Usage: Histogram Option Flags
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Contains flags to control histogram operation.

Description:

Ipi_CalcHst generates a Histogram for a channel using the Histogram Board. It takes four DVP cycles to complete and it places its results into the Histogram Buffer. If the Histogram buffer was not cleared via a call to Ipi_ClearHst then it will sum the current histogram to the one already present. Because of the size of the words in the Histogram Buffer, calculating a Histogram may cause an overflow. If this occurs, results could be unpredictable so Ipi_CalcHst returns a warning status of Ipi_HstOvrflw.

Also note that since this routine uses the DVP, it is necessary to attach and initialize it. This may be accomplished through the use of Ipi_AttDvp and Ipi_DvpInit. Between the DvpInit and the CalcHst, such DVP modifying functions such as DvpRegion, which sets a region of interest, may be called. Note, that if you specify a region of interest, you must pass the Histogram Option flag of Ipi_M_UseRegion.

Possible values for HstOps:

Flag name	If Present	If Absent
Ipi_M_UseRegion	Only Hst ROI	Hst entire image

Condition Values Returned:

SS\$_Normal	Normal successful completion
Ipi__BadParCnt	Bad parameter count
Ipi__HstOvrflw	Histogram buffer overflow

Modification History:

Date	Auth	Ver	Modification
3-Dec-1986	SLS	1.0	All new Code
27-Jan-1987	SLS	1.1	Changed over to New IpiInterface
20-Nov-1989	SLS	1.2	Removed DUP Init stuff to Ipi_DuplInit Added Region of Interest capability Changed to IPX Scheme

Authors: Stephen L. Schultz

Example:

```

Program CalcHist
*
*****      Define Unit, HstBuf, and include the Ipi Definitions
*
      Integer*4 Unit, HstBuf( 256 )
      Include 'Ipi_IpiLib'
*
*****      Attach the Unit and the DUP
*
      Call Ipi_AttUnit( Unit )
      Call Ipi_AttDvp( Unit )
*
*****      Clear the Histogram then Take one for Channel One
*
      Call Ipi_ClearHst( Unit )
      Call Ipi_DuplInit( Unit )
      Call Ipi_CalcHst( Unit, 1 )
*
*****      Get the Histogram and Print it
*
      Call Ipi_GetHst( Unit, HstBuf )
      Write (6,*) HstBuf
*
*****      Detatch the DUP and the Unit
*
      Call Ipi_DetDvp( Unit )
      Call Ipi_DetUnit( Unit )

End

```

Ipi_ClearChan - Clears an Image Channel

This routine clears an image channel via use of the DVP.

Format: Ipi_ClearChan(Unit, Channel [, Flags])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Indicates success or failure of the operation.

Arguments: Unit
Usage: Unit to Assign and Attach
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Reference
Specifies the Unit to attach the DVP to. Unit is returned by Ipi_AttUnit.

Channel
Usage: Channel to load
Type: Word (Signed)
Access: Read Only
Mechanism: By Reference
Specifies which Channel to set to a constant.

Flags
Usage: Flags to control operational [Optional]
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
These flags control how Ipi_ClearChan should clear the channel. If this argument is not present, then all the flags are assumed to be absent.

Description:

Ipi_ClearChan attaches the DVP, initializes it to a Clear opcode, executes the Clear opcode to the desired output channel, and then detaches the DVP. Note that specifying Ipi_M_ExtMem implicitly specifies Ipi_M_Sect0, 1, 2, and 3. If no flags are specified, then Ipi_M_Sect0 is implicitly specified.

Flag	If Present	If Absent
Ipi_M_ExtMem	Clear Extended Memory	Clear Standard Memory
Ipi_M_Sect0	Clear Sector 0	
Ipi_M_Sect1	Clear Sector 1	
Ipi_M_Sect2	Clear Sector 2	
Ipi_M_Sect3	Clear Sector 3	

Condition Values Returned:

SS\$_Normal Normal successful completion
Ipi_BadParCnt Bad parameter count
Any of those returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
28-Jul-1988	SLS	1.0	All new code
11-Aug-1988	SLS		Added Ipi_M_ExtMem flag
18-Nov-1988	SLS	1.1	Added Ipi_M_Sect* Flags

Authors: Stephen L. Schultz

Example:

```
Program ClearChan1k
Implicit None
Include 'Ipi_IpiLib'

Integer*4      Unit, Status, Channel, Flags, Check

Check = Ipi_M_Exit + Ipi_M_NoText
Flags = Ipi_M_ExtMem

Write (6,*) 'Enter the Channel to Clear:'
Read (5,*,End=900) Channel

Status = Ipi_AttUnit( Unit )
Call Ipi_ErrorCheck( Status, 'Error Attaching Unit',,, Check )

Status = Ipi_ClearChan( Unit, Channel, Flags )
Call Ipi_ErrorCheck( Status, 'Error Clearing Channel',,, Check )

Status = Ipi_DetUnit( Unit )
Call Ipi_ErrorCheck( Status, 'Error Detaching Unit',,, Check )

900  Continue
End
```

Ipi_ClearHst - Clear Histogram Table Buffer

This routine clears the Histogram Table Buffer which accumulates the results of a Histogram Pass.

Format: Ipi_ClearHst(Unit)

Returns: Usage: Returned Status of Operation
 Type: Longword (Unsigned)
 Access: Write Only
 Mechanism: By Value

Parameters: Unit
 Usage: Unit to Operate On
 Type: Word (Unsigned)
 Access: Read Only
 Mechanism: By Reference
 This is the unit the user will operate on. It is returned by a call to Ipi_AttUnit.

Description:

Ipi_ClearHst is used to clear out the histogram table buffer. It should be called before the user wishes to accumulate a histogram. If the user calls Ipi_ClearHst once, then calls Ipi_CalcHst three successive times, once for each channel, the resultant histogram will be a cumulative histogram representing the sum of the counts in all three channels. Where as, if the user calls Ipi_ClearHst, Ipi_CalcHst, and Ipi_GetHst, in sequence three times, once for each channel. Each histogram will represent the histogram for that specific channel.

Condition Values Returned:

SS\$_Normal Normal successful completion
Ipi__BadParCnt Bad parameter count
Any return status from Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
3-Dec-1986	SLS	1.0	All new code
27-Jan-1987	SLS	1.1	Changed to new Ipi_Qiow driver

Authors: Stephen L. Schultz

Example:

Ipi_Convolve

Perform a convolution on an image with the supplied kernel. The image MUST be located in Channel Zero. The Dvp must be attached.

Format: Ipi_Convolve(Unit, Kernel, Length, Normalize)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Arguments: Unit
Usage: Unit to work on
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the Unit to perform the convolution one. Returned by a call to Ipi_AttUnit.

Kernel
Usage: Multiplicative convolution kernel
Type: Longword Array (Signed) [Length Length^2]
Access: Read Only
Mechanism: By Reference
Kernel is an array containing the multiplicands for the convolution. The array must have a number of elements equal to the Length argument squared. If Kernel is a one dimensioned array { Fortran: Kernel(1) } instead of two dimensioned { Fortran: Kernel(X, Y) } it should be arranged as Y rows of X values, in other words, the X dimension is incremented first.

Length
Usage: Length of one dimension of the kernel
Type: Longword (Signed)
Access: Read Only
Mechanism: By Reference
Indicates the X and Y length of the kernel. The Length must be an odd value, and has a maximum value of 23.

Normalize
Usage: Boolean Flag to indicate whether to normalize
Type: Byte (Unsigned)
Access: Read Only
Mechanism: By Reference
If the Normalize flag is a zero, Ipi_Convole will not normalize its result, if the flag is a one, Ipi_Convole will normalize the result by dividing by the weight, the sum of all the elements, of the kernel.

Description:

Ipi_Convole performs a convolution on the image located in channel zero. The convolution is controlled by the supplied kernel. Where the value of a single pixel in the output image is determined by centering the kernel over the pixel, and multiplying each element in the kernel with the value of the pixel underneath it, and then summing these results and storing them at that location. This is done for every pixel in the image. Note that the results of the convolution around the edges of the image is unreliable unless the user takes measures to insure that the pixels past the edges have the proper values to convolve correctly. The result, whether

normalized or not, will be found in channel zero. If the result is not normalized, then the 16-bit result will also be located in Channel One and Two, the high and low orders respectively. Note that this routine requires: the source image in channel zero, an odd numbered length less than 23, and the DVP attached to this unit. The DVP may be attached with a call to Ipi_AttDvp.

Condition Values Returned:

SS\$_Normal	Normal successful completion
Ipi__BadParCnt	Bad parameter count
Any of those returned by Ipi_Qiow	

Modification History:

Date	Auth	Ver	Modification
07-Jun-1988	SLS	1.0	All new Code

Authors: Stephen L. Schultz

Example:

Ipi_CopyChan - Copies one Image Channel to Another

This routine uses the DVP to copy the contents of one image channel to another.

Format: Ipi_CopyChan(Unit, SrcChan, DstChan [, Flags] [, SrcFlags])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Indicates success or failure of the operation.

Arguments: Unit
Usage: Unit to Assign and Attach
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Reference
Specifies the Unit to attach the DVP to. Unit is returned by Ipi_AttUnit.

SrcChan
Usage: Source Channel
Type: Word (Signed)
Access: Read Only
Mechanism: By Reference
Specifies the Channel to be copied From.

DstChan
Usage: Destination Channel
Type: Word (Signed)
Access: Read Only
Mechanism: By Reference
Specifies the Channel to be copied To.

Flags
Usage: Flags to control operation [Optional]
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
These flags control how Ipi_CopyChan should copy the channel. If this argument is not present, then all the flags are assumed to be absent.

SrcFlags
Usage: Flags to control source operation [Optional]
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
These flags also control how Ipi_CopyChan should copy the channel. By use of these flags, the user may copy from a different source sector than the destination sector. Note that with this type of operation, only a single source sector may be specified, but multiple destination sectors may be specified, with the same source sector being copied to each destination sector. The values for SrcFlags are the same as those for Flags except that only a single sector may be specified, therefore the Ipi_M_ExtMem flag is invalid. If this argument is not present, then the routine copies from one channel to the other, with the source and destination sector being the same for each sector specified in Flags.

Description:

Ipi_CopyChan attaches the DUP, initializes it to a Nop opcode, executes the Nop to copy the source channel into the desired output channel, and then detaches the DUP. Note that specifying Ipi_M_ExtMem implicitly specifies Ipi_M_Sect0, 1, 2, and 3. If no flags are specified, then Ipi_M_Sect0 is implicitly specified

Flag	If Present	If Absent
Ipi_M_ExtMem	Load Extended Memory	Load Standard Memory
Ipi_M_Sect0	Load Sector 0	
Ipi_M_Sect1	Load Sector 1	
Ipi_M_Sect2	Load Sector 2	
Ipi_M_Sect3	Load Sector 3	

Condition Values Returned:

SS\$_Normal	Normal successful completion
Ipi__BadParCnt	Bad parameter count
Any of those returned by Ipi_Qiow	

Modification History:

Date	Auth	Ver	Modification
1-Aug-1988	SLS	1.0	All new code
11-Aug-1988	SLS		Added the Ipi_M_ExtMem flag
18-Nov-1988	SLS	1.1	Added the Ipi_M_Sect* flags
22-Nov-1988	SLS		Added the SrcFlags argument

Authors: Stephen L. Schultz

Example:

Program CopyChan1k

Implicit None
Include 'Ipi_IpiLib'

Integer*4 Unit, Status, SrcChan, DstChan, Flags, Check

Check = Ipi_M_Exit + Ipi_M_NoText
Flags = Ipi_M_ExtMem

Write (6,*) 'Enter the Channel to Copy From:'
Read (5,*,End=900) SrcChan

Write (6,*) 'Enter the Channel to Copy From:'
Read (5,*,End=900) DstChan

Status = Ipi_AttUnit(Unit)
Call Ipi_ErrorCheck(Status, 'Error Attaching Unit',,, Check)

Status = Ipi_CopyChan(Unit, SrcChan, DstChan, Flags)
Call Ipi_ErrorCheck(Status, 'Error Copying Channel',,, Check)

Status = Ipi_DetUnit(Unit)
Call Ipi_ErrorCheck(Status, 'Error Detaching Unit',,, Check)

900 Continue
End

Examples:

Ipi_DetDvp - Detatch Dvp

This subroutine detatches the Dvp from Unit the user has specified.

Format: Ipi_DetDvp(Unit)

Returns: Usage: Returned Status of Operation
 Type: Longword (Unsigned)
 Access: Write Only
 Mechanism: By Value

Parameters: Unit
 Usage: Unit to Detatch Dvp from
 Type: Word (Unsigned)
 Access: Read Only
 Mechanism: By Reference
 This is the unit the DVP is currently attatched to. This
 value is returned by a call to Ipi_AttUnit.

Description:

Ipi_DetDvp is used to free up the DVP which is a mutually exclusive device. The DVP must have been previously attached via a call to Ipi_AttDvp. Once this call to Ipi_DetDvp is made, the user may not use the DVP in any routines until he makes another call to Ipi_AttDvp.

Condition Values Returned:

SS\$_Normal Normal successfull completion
Ipi__BadParCnt Bad parameter count
Any of those returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
21-Oct-1986	SLS	1.0	All new code
27-Jan-1987	SLS	1.1	Changed over to new Ipi Interface

Authors: Stephen L. Schultz

Example:

Ipi_DvpAdvMath - Double Operand Mathematical Functions for the DVP.

This routine performs simple mathematical functions on two sets of channels and places the result into a third set. Each set of channels consists of a high order and a low order byte.

Format: Ipi_DvpAdvMath(Unit, Opcode, InChnAHi, InChnALo, InChnBHi, InChnBLo, OutChnHi, OutChnLo)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Parameters: Unit
Usage: Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the Unit to operate on. This is set by a call to Ipi_AttUnit.

Opcode
Usage: The code for the operation to perform
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Value
Specifies what operation is to be performed. These are defined as external constants. (See below)

InChnAHi
Usage: Input Channel A High Order Byte
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Defines the channel that is to be used as the first input's high order byte to the mathematical function. This input gets sent to DVP Input Bus BB.

InChnALo
Usage: Input Channel A Low Order Byte
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Defines the channel that is to be used as the first input's low order byte to the mathematical function. This input gets sent to DVP Input Bus BA.

InChnBHi
Usage: Input Channel B High Order Byte
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Defines the channel that is to be used as the second input's high order byte to the mathematical function. This input gets sent to DVP Input Bus BD.

InChnBLo
Usage: Input Channel B Low Order Byte
Type: Word (Unsigned)
Access: Read Only

Mechanism: By Reference
 Defines the channel that is to be used as the second input's low order byte to the mathematical function. This input gets sent to DVP Input Bus BC.

OutChnHi
 Usage: Output Channel C High Order Byte
 Type: Word (Unsigned)
 Access: Read Only
 Mechanism: By Reference
 Defines the channel that is to be used as the output's high order byte from the mathematical function. This output come from DVP Output Channel DvpB.

OutChnLo
 Usage: Output Channel C Low Order Byte
 Type: Word (Unsigned)
 Access: Read Only
 Mechanism: By Reference
 Defines the channel that is to be used as the output's low order byte from the mathematical function. This output come from DVP Output Channel DvpA.

Description:

Ipi_DvpMath sets the 1st Alu Opcodes according to the function specified by the user. The 1st Alus are set up to operate as 16 bit opcodes. It is important that the DVP is set up properly prior to calling this routine. This can be done by a call to Ipi_DvpInit just prior to calling this routine. The DVP Input Buses each input channel is connected to is listed above under the description of the parameter in the event that the user wishes to assign constants to the inputs. This must be done by a call to Ipi_DvpConstant between the Ipi_DvpInit and the Ipi_DvpAdvMath.

The following is a list of the current DvpAdvMath Opcodes, their assigned value, and their associated function:

Ipi_Minus	002	C = A - B
Ipi_Plus	003	C = A + B
Ipi_Not	011	C = .Not. A (1's compliment)

In a Fortran program, simply define the opcode as an external integer function. This will ensure that the linker picks up the symbol and that Fortran will pass it By Value to this routine. If you use the following include statement:

```
INCLUDE 'IPI_IPILIB'
```

The opcode definitions are automatically done for you. All you need do is use them.

Condition Values Returned:

SS\$_Normal	Normal successful completion
Ipi_BadParCnt	Bad parameter count
Ipi_BadMthOpc	Bad math opcode
Any of those returned by Ipi_Qiow	

Modification History:

Date	Auth	Ver	Modification
22-Jan-1987	SLS	1.0	All new Code (Now in Ipi_Multiply)
27-Jan-1987	SLS	1.1	Changed to new Driver Interface
24-Aug-1987	SLS	1.2	Totally new rewrite
18-Jul-1989	SLS		Added Ipi_Not

Authors: Stephen L. Schultz

Example:

```

      Program SumChannel
*
*      Sums one set of channels to the other. Each set is a
*      16 bit number as follows: [2,3] = [2,3] + [0,1]
*
*****      Define Unit, and include the Ipi Definitions
*
      Integer*4 Unit, Status
      Include 'Ipi_IpiLib'
*
*****      Attach the Unit and the Dvp ( with Wait )
*
      Status = Ipi_AttUnit( Unit )
      Call Ipi_ErrorCheck( Status, 'Error in AttUnit' )

      Call Ipi_AttDvp( Unit, 1 )
*
*****      Initialize the DVP then perform the Addition
*
      Call Ipi_DvpInit( Unit )
      Call Ipi_DvpMath( Unit, Ipi__Add, 0, 1, 2, 3, 2, 3 )
*
*****      Detach the Dvp and Unit
*
      Call Ipi_DetDvp( Unit )
      Call Ipi_DetUnit( Unit )
*
*****      End of Program SumChannel
*
      End
```


Ipi_DvpConstant - Sets the Value for a DVP Constant

This routine allows the user to set the value for a DVP constant.
This routine should be called after a call to Ipi_DvpInit.

Format: Ipi_DvpConstant(Unit, [ConstA] [, ConstB] [, ConstC]
[. . .] [, ConstL])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Completion status of the Operation

Arguments: Unit
Usage: The DeAnza Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
The Unit Channel Number to perform the operation on.
This value is returned by a call to Ipi_AttUnit.

ConstA, ConstB, ConstC, . . . , ConstL
Usage: The Value for Constant A (B, C, etc.)
Type: Byte
Access: Read Only
Mechanism: By Reference
Contains the value for the constant that is to be set.
If the constant value is not there, the constant will
not be used in DVP processing.

Description:

Ipi_DvpConstant allows the user to Initialize the value of
a Dvp Constant and use it in place of the corresponding Dvp
input. Note that the commas must be included to separate any
of the optional constants. So Ipi_DvpConstant(Unit,,, ConstC)
would set only Constant C. Also note that there is no ConstG
or ConstI. Don't ask me why, it was DeAnza's decision.

Condition Values Returned:

SS\$_Normal Normal successful completion
Ipi_BadParCnt Bad parameter count
Any of those returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
29-Jul-1987	SLS	1.0	All new Code (was Ipi_SetConstant)
28-Jul-1988	SLS	1.1	Renamed to Ipi_DvpConstant

Authors: Stephen L. Schultz

Example:

```
***** This Routine will set the Blue Screen to all Blue
*
Call Ipi_AttUnit( Unit )
Call Ipi_AttDvp( Unit )
Call Ipi_DvpInit( Unit )
Call Ipi_DvpConstant( Unit, 255 )
Call Ipi_DvpMath( Unit, Ipi_Nop, 2, 2, 2 )
Call Ipi_DetDvp( Unit )
Call Ipi_DetUnit( Unit )
End
```

Ipi_DvpGo - Activate the DVP

This routine activates the DVP.

Format: Ipi_DvpGo(Unit)

Returns: Usage: Returned Status of Operation
 Type: Longword (Unsigned)
 Access: Write Only
 Mechanism: By Value

Parameters: Unit
 Usage: Unit to Operate On
 Type: Longword (Unsigned)
 Access: Read Only
 Mechanism: By Reference
 Specifies the Unit to operate on. This is set by a call
 to Ipi_AttUnit.

Description:

Ipi_DvpGo activates the DVP. In order for this routine to function, the user must have previously attached the DVP via a call to Ipi_AttDvp, and assigned input and output buses, the most common method of doing so being calls to Ipi_DvpInput and Ipi_DvpOutput.

Condition Values Returned:

SS\$_Normal Normal successful completion
Ipi_BadParCnt Bad parameter count
Any of those returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
1-Jun-1988	SLS	1.0	All new Code

Authors: Stephen L. Schultz

Example:

Ipi_DvpInput - Assign Channels to the DVP Input Buses.

This routine allows the user to explicitly attach channels to the DUP Input Buses. Note that some routines, such as `Ipi_DvpMath` and `Ipi_DvpAdvMath`, implicitly assign input channels and therefore, override these assignments for any channels they use.

[illegible]

Returns:	Usage:	Returned Status of Operation
	Type:	Longword (Unsigned)
	Access:	Write Only
	Mechanism:	By Value

Parameters:	Unit
Usage:	Unit to Operate On
Type:	Longword (Unsigned)
Access:	Read Only
Mechanism:	By Reference
	Specifies the Unit to operate on. This is set by a call to Ipi_AttUnit.

```
BA, BB, BC, BD, BE, BF, BH, BJ
Usage:      DVP Input Bus
Type:       Word (Unsigned)
Access:     Read Only
Mechanism:  By Value
Indicates the input bus you want the channel attached to.
Any or all of them are optional, but positions must be
preserved by commas if latter ones are present when prior
ones are not.
```

Description:

`Ipi_DvpInput` is used to explicitly assign the DVP Input Buses. It should be called after `Ipi_DvpInit` but before a call to an operation that activates the DVP. A single channel may be assigned to multiple DVP Input Buses.

Condition Values Returned:

SS\$ _Normal	Normal successful completion
lpi_BadParCnt	Bad parameter count
Any of those returned by lpi_Qiow	

Modification History:

Date	Auth	Ver	Modification
1-Jun-1988	SLS	1.0	All new Code

Authors: Stephen L. Schultz

Example:

Ipi_DvpInit - Initialize Digital Video Processor

This routine initializes the DVP to a known state. The state used is the most common settings for the various DVP registers.

Format: Ipi_DvpInit(Unit)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Arguments: Unit
Usage: Unit to perform the operation on
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Reference
Specifies the Unit through which to initialize the DVP.

Description:

Ipi_DvpInit is used to initialize the DVP for the various routines that use the DVP such as Ipi_DvpMath, Ipi_DvpAdvMath, Ipi_DvpClip, Ipi_DvpMinMax and Ipi_CalcHst. After calling Ipi_DvpInit, the user may explicitly connect input buses and output buses using Ipi_DvpInput and Ipi_DvpOutput, or implicitly connect input and output buses via the supplied parameters in Ipi_DvpMath and Ipi_DvpAdvMath.

Ipi_DvpInit sets the DVP up in following manner:

- Select Dvp Input Buses (No Constants)
- Bypass all Multipliers (PA=BA, PB=BB, PC=BC, PD=BD, etc)
- Direct connect to First Alu Inputs (FIA=PA, FIB=PC, FIC=PB, FID=PD, FIE=PE, FIF=PH, FIH=PF, FIJ=PJ)
- 1st Alu Opcode: Output = Input A, with write select
- Direct connect all 2nd Alu Input A's (SIA=FOA, SIC=FIB, SIE=FOC, SIH=FOD)
- Zero connect all 2nd Alu Input B's (SIB=ZG, SID=ZG, SIF=ZG, SIJ=ZG)
- 2nd Alu Opcode: Output = Input A + Input B
- Direct connect all output buses (DIPA=SOA, DIPB=SOB, DIPC=SOC, DIPD=SOD)
- 1st Alu's are in four 8-bit mode.
- All control word logic is masked, only first entry is used.
- Control word operation is Output = Input A
- No Accumulation, Comparisons, Clipping, etcetera, is done.
- No Shifting is done.
- Region of Interest set between [0,511] and [511,0]
- All constants set to Zero.
- Sets all bit plane masks to all 1's.

If the DVP is activated without changing any of these then the following paths are created:

DIPA = BA, DIPB = BB, DIPC = BE, DIPD = BF

Note that if you do not use one of the canned routines, you must make your own bus assignments to these input and output DVP Buses.

Condition Values Returned:

SS\$_Normal

Normal successful completion

lpi_BadParCnt Bad parameter count
Any of those returned by lpi_Qiow

Modification History:

Date	Auth	Ver	Modification
11-Jun-1987	SLS	1.0	All new Code
29-Jul-1987	SLS	1.1	Initialized AP Destination Regs
05-Oct-1989	SLS	1.2	Converted to IPX scheme
15-Feb-1990	SLS	1.3	Set bit plane masks to all 1's

Authors: Stephen L. Schultz

Example:

Ipi_DvpMath - Simple Mathematical Functions for the DUP

This routine performs simple mathematical functions on two channels and places the result into a third.

Format: Ipi_DvpMath(Unit, Opcode, InChnA, InChnB, OutChn)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Parameters: Unit
Usage: Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the Unit to operate on. This is set by a call to Ipi_AttUnit.

Opcode
Usage: The code for the operation to perform
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Value
Specifies what operation is to be performed. These are defined as external constants.

InChnA
Usage: Input Channel A
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Defines the channel that is to be used as the first input to the mathematical function.

InChnB
Usage: Input Channel B
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Defines the channel that is to be used as the second input to the mathematical function.

OutChn
Usage: Output Channel C
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Defines the channel that is to be used as the output from the mathematical function.

Description:

Ipi_DvpMath sets the 2nd Alu Opcode to perform the mathematical function specified by the user. It is important that the DUP is set up properly prior to calling this routine. This can be done by a call to Ipi_DvpInit just prior to calling this routine. The following is a list of the current DvpMath Opcodes, their assigned value, and their associated function:

Ipi_Clear	001	C = 0
Ipi_Minus	002	C = A - B

Ipi_Plus	003	C = A + B
Ipi_Xor	004	C = A .xor. B
Ipi_Or	005	C = A .or. B
Ipi_And	006	C = A .and. B
Ipi_Set	007	C = 255
Ipi_Nop	008	C = A

Condition Values Returned:

SS\$_Normal	Normal successful completion
Ipi_BadParCnt	Bad parameter count
Ipi_BadMthOpc	Bad math opcode
Any of those returned by Ipi_Qiow	

Modification History:

Date	Auth	Ver	Modification
22-Jan-1987	SLS	1.0	All new Code
27-Jan-1987	SLS	1.1	Changed over to New IpiInterface
10-Apr-1987	SLS	1.2	Enabled/Disabled Dvp to Mem writing bit
11-Jun-1987	SLS	1.3	Relocated sections to Ipi_DvpInit

Authors: Stephen L. Schultz

Example:

```

      Program AddChannels
*
*****      Define Unit, and include the Ipi Definitions
*
      Integer*4 Unit, Status
      Include 'Ipi_IpiLib'
*
*****      Attach the Unit and the Dvp ( with Wait )
*
      Status = Ipi_AttUnit( Unit )
      Call Ipi_ErrorCheck( Status, 'Error in AttUnit' )

      Call Ipi_AttDvp( Unit, 1 )
*
*****      Initialize the DVP then perform the Addition
*
      Call Ipi_DvpInit( Unit )
      Call Ipi_DvpMath( Unit, Ipi_Add, 0, 1, 2 )
*
*****      Detach the Dvp and Unit
*
      Call Ipi_DetDvp( Unit )
      Call Ipi_DetUnit( Unit )
*
*****      End of Program AddChannel
*
      End

```

Ipi_DvpMinMax - Takes the Minimum or Maximum of Two Channels

This routine allows the user to take the Minimum or the Maximum of two channels on a per pixel basis. It may not be used in conjunction with Ipi_DvpClip.

Format: Ipi_DvpMinMax(Unit, Opcode, Mode)

Returns:

Usage:	Returned Status of Operation
Type:	Longword (Unsigned)
Access:	Write Only
Mechanism:	By Value

Arguments:

Unit

Usage:	Unit to perform the operation on
Type:	Longword (Unsigned)
Access:	Write Only
Mechanism:	By Reference

Specifies the Unit through which to access the DUP.

Opcode

Usage:	The code for the operation to perform
Type:	Word (Unsigned)
Access:	Read Only
Mechanism:	By Value

Specifies what operation is to be performed. These are defined as external constants.

Mode

Usage:	Mode to Clip in.
Type:	Word (Unsigned)
Access:	Read Only
Mechanism:	By Reference

Specifies whether to clip in Signed or Unsigned and 8-bit or 16-bit.

Description:

Ipi_DvpMinMax is used to produce the minimum or maximum of two image channels on a per pixel basis. Because this uses the DUP Comparator, this may not be used in conjunction with Ipi_DvpClip, which also use the comparator. In 8-bit mode, the input comes from SOA for one source and SIE for the other source, or if data paths have been unaltered since a call to Ipi_DvpInit, it will be whatever is supplied to the DUP BA and BE Buses respectively. In 16-bit mode, the input is from SOA and SOB supplying the low and high order bytes respectively for one source, and SIE and SIH supplying the low and high order bytes for the other source. Again, in an unaltered data path, those those correspond to inputs BA and BB, and BE and BF, respectively. The input buses may be assigned using Ipi_DvpInput. The output buses may be assigned using Ipi_DvpOutput.

The values for the Mode are:

- | | |
|---|-------------------|
| 0 | - 8-Bit Unsigned |
| 1 | - 8-bit Signed |
| 2 | - 16-bit Unsigned |
| 3 | - 16-bit Signed |

The values for the Opcode are:

- | | |
|-------------|--------------------|
| Ipi_Minimum | - Take the Minimum |
| Ipi_Maximum | - Take the Maximum |

Condition Values Returned:

SS\$_Normal	Normal successful completion
-------------	------------------------------

Ipi_BadParCnt Bad parameter count
Any of those returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
1-Jun-1988	SLS	1.0	All new Code

Authors: Stephen L. Schultz

Example:

Ipi_DvpMultiply - Multiplies two 8-Bit channels

This routine multiplies two 8-bit channels producing a 16-bit result.

Format: Ipi_DvpMultiply(Unit, InChnA, InChnB, OutChnHi, OutChnLo)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
The completion status of the operation.

Parameters: Unit
Usage: Unit to Operate On
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
The unit to which the DVP is attached. This value is returned via a call to Ipi_AttUnit.

InChnA
Usage: Input Channel A
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
The first of two input channels to multiply.

InChnB
Usage: Input Channel B
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
The second of two input channels to multiply.

OutChnHi
Usage: The Output High Order Channel
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
The Output Channel that will receive the high order byte of the result.

OutChnLo
Usage: The Output Low Order Channel
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
The Output Channel that will receive the low order byte of the result.

Description:

Ipi_Multiply is used to multiply two 8-bit numbers producing a 16-bit result. The 16-bit result is sent to two channels. The high order (most significant) byte is sent to the output channel indicated by OutChnHi, and the low order (least significant) byte is sent to the output channel indicated by OutChnLo. Before this routine can be successfully called, a call to Ipi_DvpInit must be made.

Condition Values Returned:
SS\$_Normal Normal successful completion

Ipi__BadParCnt Bad parameter count
Any returned by Ipi_Qiow

Modification History:

22-Jan-1987	SLS	.*	All new code (In Ipi_DvpAdvMath)
27-Jan-1987	SLS	.*	Converted to new Driver Interface
24-Aug-1987	SLS	1.0	Moved Multiply to its own routine

Authors: Stephen L. Schultz

Example:

```
Implicit None
Include 'Ipi_IpiLib'

Integer*4            Unit, A, B, CHi, CLo

Write (6,*) 'Input A, B, CHi, CLo (A x B = [CHi,CLo]) : '
Read (5,*,End=900) A, B, CHi, CLo

Call Ipx_AttUnit( Unit )
Call Ipx_AttDvp( Unit, 1 )
Call Ipx_DvpInit( Unit )
Call Ipx_DvpMultiply( Unit, A, B, CHi, CLo )
Call Ipx_DetDvp( Unit )
Call Ipx_DetUnit( Unit )

900    Continue
      End
```

Ipi_DvpOutput - Assign / Deassign Channels to the DVP Output Buses.

This routine allows the user to explicitly attach channels to the DVP Output Buses. Note that some routines, such as Ipi_DvpMath and Ipi_DvpAdvMath, implicitly assign Output channels and therefore, override these assignments for any channels they use.

Format: Ipi_DvpOutput(Unit, [Bus], Chan [, Chan . . .])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Parameters: Unit
Usage: Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the Unit to operate on. This is set by a call to Ipi_AttUnit.

Bus
Usage: DVP Output Bus
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Indicates the Output bus you want the channel attached to. If this argument is not present, then all the channels indicated will be deassigned from output buses.

Chan
Usage: Channel to assign to that bus
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Indicates the channel(s) you want assigned to this output bus. There can be any number of these arguments.

Description:

Ipi_DvpOutput is used to explicitly assign the DVP Output Buses. It should be called after Ipi_DvpInit but before a call to an operation that activates the DVP. A single channel may only be assigned to one output bus, however, each output bus may have be assigned to multiple channels. Multiple calls to this routine will assign multiple buses. Note that a second attempt to assign the same channel will result in only the second assignment taking place. Because of the nature of this multiple assignment, after the DVP operation is complete, the user should then deassign all output buses. This can be done by leaving out the Bus argument. Note that the comma must still be there to hold its place. The valid values for Bus are:

- 0 - Output Bus DvpA
- 1 - Output Bus DvpB
- 2 - Output Bus DvpC
- 3 - Output Bus DvpD

Condition Values Returned:

SS\$_Normal Normal successful completion
Ipi_BadParCnt Bad parameter count

Any of those returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
1-Jun-1988	SLS	1.0	All new Code

Authors: Stephen L. Schultz

Example:

Ipi_DvpRegion - Sets the Region of Interest for the DVP

This routine allows the user to define the Region of Interest for the DVP routines. This routine should be called after a call to Ipi_DvpInit.

Format: Ipi_DvpRegion(Unit, StartX, StartY, EndX, EndY)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Completion status of the Operation

Arguments: Unit
Usage: The DeAnza Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
The Unit Channel Number to perform the operation on.
This value is returned by a call to Ipi_AttUnit.

StartX
Usage: The Starting X Coordinate
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Indicates the Start of the Region of Interest in the X Direction.

StartY
Usage: The Starting Y Coordinate
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Indicates the Start of the Region of Interest in the Y Direction.

EndX
Usage: The Ending X Coordinate
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Indicates the End of the Region of Interest in the X Direction.

EndY
Usage: The Ending Y Coordinate
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Indicates the End of the Region of Interest in the Y Direction.

Description:

Ipi_DvpRegion allows the user to define the Region of Interest for DVP Operations. Ipi_DvpInit initializes the Region of Interest to be from [0,0] to [511,511], or in other words, the entire low resolution screen.

Right now (version 1.2) there are some problems when trying to

use Ipi_DvpRegion with some of the higher order DVP functions. Specifically ones that use the DUP's First ALU to perform their functionality. The reason as that Ipi_DvpRegion must modify the ROI mask and opcodes. Because the Control Word Register is read only, there is no way to tell if ROI processing has been enabled or not.

Condition Values Returned:

SS\$_Normal	Normal successful completion
Ipi__BadParCnt	Bad parameter count
Any of those returned by Ipi_Qiow	

Modification History:

Date	Auth	Ver	Modification
29-Jul-1987	SLS	1.0	All new Code (was Ipi_SetRegion)
28-Jul-1988	SLS	1.1	Renamed to Ipi_DvpRegion
20-Nov-1989	SLS	1.2	Added CCMNMX Bit Enable for inside ROI Converted to IPX scheme

Authors: Stephen L. Schultz

Example:

```

*****      This Routine will set a portion Blue Screen to all Blue
*
      Call Ipi_AttUnit( Unit )
      Call Ipi_AttDvp( Unit )
      Call Ipi_DvpInit( Unit )
      Call Ipi_DvpRegion( Unit, 100, 100, 200, 300 )
      Call Ipi_DvpConstant( Unit, 255 )
      Call Ipi_DvpMath( Unit, Ipi_Nop, 2, 2, 2 )
      Call Ipi_DetDvp( Unit )
      Call Ipi_DetUnit( Unit )

      End

```

Ipi_EnableItt - Enable Itt Transfers

This routine allows the user to enable the Itts either between Memory and the Vocs or between Memory and the DUP.

Format: Ipi_EnableItt(Unit, Chan, MemVoc, MemDvp [, ChnMsk])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value

Parameters: Unit
Usage: Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
This is the unit that the operation is to be performed on.
This value is returned by a call to Ipi_AttnUnit.

Chan
Usage: Channel to Enable Itt on
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
This represents the Channel to Enable/Disable Itt Processing on. If the Channel Mask Operand is present, this operand is used during any Ipi_Drv_Rmc operations to determine the current settings of the various bits.

MemVoc
Usage: Indicates Enabling Between Mem and Voc
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
This flag is used to determine whether the Itt is Enabled or Disabled between Memory and the VOC. A Zero represents Disabling and a One represents Enabling.

MemDvp
Usage: Indicates Enabling Between Mem and Dvp
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
This flag is used to determine whether the Itt is Enabled or Disabled between Memory and the DUP. A Zero represents Disabling and a One represents Enabling.

ChnMsk
Usage: Channel Mask to Enable Itt on (Optional)
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
This optional argument is used to perform the same operation to multiple channels. Each channel to operate on is indicated by its corresponding bit being set.

Description:

Ipi_EnableItt is used to enable or disable Itt processing. It may be used on either a single channel, or may be broadcast to multiple channels depending on whether the channel mask argument is present

or not. There are two places ltt Processing may take place. The first is between the Memory and the VOC. Enabling ltt processing here has the result of modifying the way the image is viewed on the monitor. The second is between Memory and the DVP. This has the result of modifying the input to the DVP. One method for performing quick mathematical functions is to load the corresponding values for the function into the ltt's, enabling ltt processing between the DVP and Memory, and using the lpi__Nop mathematical operation in lpi_DvpMath. This will "pass" the image through the ltt saving it back to memory.

Condition Values Returned:

SS\$_Normal Normal successfull completion
 lpi__BadParCnt Bad parameter count
 Any of those returned by lpi_Qiow

Modification History:

Date	Auth	Ver	Modification
21-Oct-1986	SLS	1.0	All new code
27-Jan-1987	SLS	1.1	Changed to new lpi Interface
29-Apr-1987	SLS	1.2	Added Channel Mask Broadcast capability

Authors: Stephen L. Schultz

Example:

Description:

Ipi_ErrorCheck is a convenient way to inform the user of the success or failure of the Ipi call whose status is supplied. If it was unsuccessful it will indicate an error. If used in a Fortran If statement, it will also allow the programmer to conditionally branch depending on the success or failure of the operation.

The possible values for Flags are:

Ipi_M_Force	- Forces printing of the error message
Ipi_M_Exit	- Calls Sys\$Exit on errors
Ipi_M_NoText	- Don't print the error message

Condition Values Returned:

None

Modification History:

Date	Auth	Ver	Modification
28-Oct-1986	SLS	1.0	All new code
21-May-1987	SLS	1.1	Added Smg\$Put_Line capability
08-Feb-1988	SLS	1.2	Added Force parameter
29-Jul-1988	SLS	1.3	Changed Force parameter to Flags

Authors: Stephen L. Schultz

Example:

Ipi_GetHst - Get the Histogram Table Buffer

This routine reads the Histogram Table Buffer which contains the accumulation of any Histogram Calculations since the last time the buffer was cleared.

Format: Ipi_GetHst(Unit, HstBuf)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Returned status of the operation.

Parameters: Unit
Usage: Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
The Unit to read the Histogram through. This value is returned through a call to Ipi_AttUnit.

HstBuf
Usage: Buffer to receive histogram values
Type: Longword (Signed) Buffer [Length 256]
Access: Read Only
Mechanism: By Reference
The buffer to receive the contents of the Histogram Table Buffer.

Description:

This call retrieves the values accumulated in the Histogram Table Buffer. The buffer is not cleared after a call to Ipi_GetHst. In order to clear it, the user must make a call to Ipi_ClearHst. The Histogram Table Buffer gets accumulated by making a call to Ipi_CalcHst.

Condition Values Returned:

SS\$_Normal Normal successful completion
Ipi__BadParCnt Bad parameter count
Any returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
3-Dec-1986	SLS	1.0	All new code
27-Jan-1987	SLS	1.1	Changed to new driver interface

Authors: Stephen L. Schultz

Example:

Ipi_GetPic - Reads a picture from Image Memory

This routine reads an image from the specified image memory to a supplied buffer. The manner in which it reads is determined by PicOps.

Format: Ipi_GetPic(Unit, ChnMsk, PicBuf, PicOps)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Indicates success or failure of the operation.

Parameter: Unit
Usage: Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
This is the channel for the DeAnza. It is returned by a call to Ipi_AttUnit.

ChnMsk
Usage: Channels to read picture to
Type: Longword (Unsigned) [Array Length N]
Access: Read Only
Mechanism: By Reference
A channel mask is used to indicate which channels to read to. Each bit in the 32-bit mask corresponds to a channel on the Unit. If the bit is set, data will be sent to that channel. If this is a color or multiband image, ChnMsk must be an array with one element in the array for each band in the image (3 for a color image). If the user specified Ipi_M_WordData, that doubles the number of necessary ChnMsk elements, with each pair of elements representing the High and Low order bits of the single word of data.

PicBuf
Usage: Buffer of image values
Type: Byte (Unsigned) Array [Length R x R]
Access: Write Only
Mechanism: By Reference
This array will receive the actual image data. The size of the array depends on the X and Y Dimensions as set in the PicOps. Also, if Ipi_M_WordData is specified, then this becomes a Word (Unsigned) Array.

PicOps
Usage: Picture Options Block
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the address of the Picture Options Block that is defined by Ipi_SetPicOps and related calls.

Description:

Ipi_GetPic will load an image from the screen according to the settings in PicOps. A standard monochrome, 512x512 image would be loaded from standard image memory with the first line in the file being read from the top of the screen, down to the last line in the file being read from the bottom of the screen. The direction

of loading may be modified by the following PicOps settings:

- Ipi_M_XYSwitch - Switches the X and Y axis in effect rotating the image about the line $Y = -X$
- Ipi_M_XInvert - Loads the X Axis in backwards
- Ipi_M_YInvert - Loads the Y Axis in backwards

The position on the screen and size of the image on the screen may be modified by other PicOps routines:

- Call Ipi_SetPicSize - Defines the X and Y Size of the image on the screen.
- Call Ipi_SetPicLocation - Defines the X and Y Location of the image on the screen.

*Note: Ipi_M_ExtMem must be set in the PicOps if you setting the Size or Location would cause the image to extend beyond the 512 x 512 standard memory.

The data window within the image file may be modified by the following PicOps routines:

- Call Ipi_SetPicDimension - Defines the X and Y Size of the image in the file. Only the portion corresponding to X and Y Size will be displayed.
- Call Ipi_SetPicOffset - Defines the X and Y Offset within the image file that the displayed portion will be removed from.

By default, the Size and Dimension of an image are the same, setting the size will automatically set the dimension, unless the dimension was explicitly changed. Also, by default, the Offsets are both zero, and the Location is the lower left corner of the display corresponding to 0 for the X and 0 for the Y.

Other settings that modify how the image is read:

- Ipi_M_WordData - Each pixel is represented, and read as 16-bits instead of 8-bits. This requires the user to specify twice the number of Channel Masks normally required for the image since now each mask must instead be a pair of masks, each pair consisting of the High and Low order bytes. So a Color Word image would require 6 channel masks.
- Ipi_M_Color - Each pixel is represented by three discrete elements, a Red, Green, and Blue element. There must be one channel mask for each element.
- Call Ipi_SetPicBands - This can set the explicit number of bands for a multiband image. This should not be used in conjunction with the Ipi_M_Color option. There must be one channel mask for each band.

Condition Values Returned:

- Ipi_BadParCnt - Bad parameter count
- Any of those returned by Ipi_Qiow.

Modification History:

Date	Auth	Ver	Modification
14-Nov-1986	SLS	1.0	All new code
11-Feb-1987	SLS		Cleaned up some transfer variables
03-Mar-1987	SLS	1.1	Added Band Interleaving
06-Apr-1987	SLS		Added Line Interleaving
29-May-1987	SLS	1.2	Rewrite for full options

15-Feb-1988	SLS	1.3	Skip bands with an empty channel mask
04-Aug-1988	SLS	1.4	Rewrote for new PicOps scheme
18-Jan-1989	SLS	1.5	Converted to IPX scheme
24-Jul-1990	SLS	1.6	Adjusted transfer for small image sizes

Ipi_ImageVector - Write a Vector to Image Memory

This routine writes a Vector to the specified Image Channels.

Format: Ipi_ImageVector(Unit, ChnMsk, Intens, Count, Points)

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Returns the status of the operation.

Parameters: Unit
Usage: Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Unit whose channels to write to. This value is set by a call to Ipi_AttUnit.

ChnMsk
Usage: Channel Mask to write Vector with
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
A channel mask which indicates which channels to write the vector to.

Count
Usage: Number of Vectors
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
A count indicating the number of vectors to write.

Intens
Usage: Intensity to Write Vectors at
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
The pixel intensity to write the vector at.

Points
Usage: Vector Start and End Points in groups of 2
Type: Word (Unsigned) [Length (2 + (Count x 2))]
Access: Read Only
Mechanism: By Reference
The X and Y Pairs for the vector end points. Note that you need to supply an X and Y Start Points for each call plus one X and Y End Point for each successive vector (ie: an X Y pair plus an X Y pair per Count)

Description:

Ipi_ImageVector is used to draw vectors to the image channels. It actually sets the values in the image memory. A vector, or set of vectors, is defined by an X and Y starting point and an X and Y ending point. For each additional vector in the call, you need only specify the X and Y ending point since it will draw from the previous ending point. So to draw a 100 pixel square starting at [200,200], the Points array would contain the following:

Points(1) = 200	Points(2) = 200
Points(3) = 300	Points(4) = 200
Points(5) = 300	Points(6) = 300
Points(7) = 200	Points(8) = 300
Points(9) = 200	Points(10) = 200

Count would equal 4 since four vectors are being drawn

Condition Values Returned:

SS\$Normal	Normal successfull completion
Ipi_BadParCnt	Bad parameter count

Any of those returned by Ipi_Qiow.

Modification History:

Date	Auth	Ver	Modification
25-Feb-1987	SLS	1.0	All new code

Authors: Stephen L. Schultz

Example:

Ipi_SavePic - Saves a Picture from Image Memory to Disk

This routine saves a picture from image memory to disk according to the settings in the PicOps block.

```
Format:      Ipi_SavePic( Unit, Filename, ChanMask, PicOps
                [ , Default ] [ , Resultant ] )
```

Returns:	Usage:	Returned Status of Operation
	Type:	Longword (Unsigned)
	Access:	Write Only
	Mechanism:	By Value
		Indicates success or failure of the operation.

```
Arguments:    Unit
Usage:        Unit to Assign and Attach
Type:         Longword (Unsigned)
Access:       Write Only
Mechanism:    By Reference
Specifies the Unit to attach the DVP to.  Unit is returned
by Ipi_AttUnit.
```

```

FileName
Usage:      Name of the Picture File
Type:      Character Array
Access:     Read Only
Mechanism:  By Descriptor
Contains the name of the file to save the picture to.
It is passed to the File-Spec parameter of Lib$Find_File
to locate the actual file.

```

```
ChannelMask
Usage:      Mask of Channels
Type:      Longword (Unsigned) [Array Length N]
Access:    Read Only
Mechanism: By Reference

A channel mask is used to indicate which channels to read to.
Each bit in the 32-bit mask corresponds to a channel on the
Unit.  If the bit is set, data will be sent to that channel.
If this is a color or multiband image, ChnMsk must be an array
with one element in the array for each band in the image ( 3
for a color image ).  If the user specified Ipi_M_WordData,
that doubles the number of necessary ChnMsk elements, with
each pair of elements representing the High and Low order
bits of the single word of data.
```

PicOps	
Usage:	Picture Options Block
Type:	Longword (Unsigned)
Access:	Read Only
Mechanism:	By Reference
Specifies the address of the Picture Options Block that is defined by Ipi_SetPicOps and related calls.	

```

Default
Usage:      Default File Spec [ Optional ]
Type:       Character Array
Access:     Read Only
Mechanism:  By Descriptor
Contains any default specifications for the file name.
It is passed to the Default-Spec parameter of Lib$Find_File

```

to locate the actual file.

Resultant

Usage: Resultant File Spec [Optional]

Type: Character Array

Access: Modify

Mechanism: By Descriptor

Contains any related file specifications for the file name. After parsing the filename, this will contain the resulting file specification that was actually located by Lib\$Find_File. It is passed to the Related-Spec parameter of Lib\$Find_File to locate the actual file, and then the Result-Spec parameter from Lib\$Find_File is copied into it upon completion.

Description:

Ipi_SavePic will read an image from the screen according to the settings in PicOps. A standard monochrome, 512x512 image would be loaded from standard image memory with the first line in the file being read from the top of the screen, down to the last line in the file being read from the bottom of the screen. The direction of loading may be modified by the following PicOps settings:

- Ipi_M_XYSwitch - Switches the X and Y axis in effect rotating the image about the line Y = -X
- Ipi_M_XInvert - Loads the X Axis in backwards
- Ipi_M_YInvert - Loads the Y Axis in backwards

The position on the screen and size of the image on the screen may be modified by other PicOps routines:

- Call Ipi_SetPicSize - Defines the X and Y Size of the image on the screen.
- Call Ipi_SetPicLocation - Defines the X and Y Location of the image on the screen.

*Note: Ipi_M_ExtMem must be set in the PicOps if you setting the Size or Location would cause the image to extend beyond the 512 x 512 standard memory.

The data window within the image file may be modified by the following PicOps routines:

- Call Ipi_SetPicDimension - Defines the X and Y Size of the image in the file. Only the portion corresponding to X and Y Size will be displayed.
- Call Ipi_SetPicOffset - Defines the X and Y Offset within the image file that the displayed portion will be removed from.

By default, the Size and Dimension of an image are the same, setting the size will automatically set the dimension, unless the dimension was explicitly changed. Also, by default, the Offsets are both zero, and the Location is the lower left corner of the display corresponding to 0 for the X and 0 for the Y.

Other settings that modify how the image is read:

- Ipi_M_WordData - Each pixel is represented, and read as 16-bits instead of 8-bits. This requires the user to specify twice the number of Channel Masks normally required for the image since now each mask must instead be a pair of masks, each pair consisting of the High and Low order bytes. So a Color Word image would require 6 channel masks.
- Ipi_M_Color - Each pixel is represented by three discrete elements, a Red, Green, and Blue element. There must be one channel mask for each element.

Call Ipi_SetPicBands - This can set the explicit number of bands for a multiband image. This should not be used in conjunction with the Ipi_M_Color option. There must be one channel mask for each band.

Condition Values Returned:

SS\$_Normal Normal successful completion
 Ipi_BadParCnt Bad parameter count
 Any of those returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
05-Aug-1988	SLS	1.0	All new code

Authors: Stephen L. Schultz

Example:

Program SaveColor

Implicit None
 Include 'Ipi_IpiLib'

Integer*4 Unit, Status, ChanMask, PicOps, Check
 Character*40 Filename, Default

Check = Ipi_M_Exit + Ipi_M_NoText
 Default = '*.Pix'

Write (6,*) 'Enter the Filename of the Color Image:'
 Read (5,'(a40)',End=900) Filename

Write (6,*) 'Enter the Red, Green, and Blue Channel Masks:'
 Read (5,*,End=900) ChanMask(1), ChanMask(2), ChanMask(3)

Status = Ipi_AttUnit(Unit)
 Call Ipi_ErrorCheck(Status, 'Error Attaching Unit',,, Check)

Status = Ipi_SetPicOps(PicOps, Ipi_M_Color)
 Call Ipi_ErrorCheck(Status, 'Error Setting Options',,, Check)

Status = Ipi_SavePic(Unit, Filename, ChanMask, PicOps, Default)
 Call Ipi_ErrorCheck(Status, 'Error Saving Image',,, Check)

Status = Ipi_DetUnit(Unit)
 Call Ipi_ErrorCheck(Status, 'Error Detaching Unit',,, Check)

900 Continue
 End

Ipi_Scroll - Scroll image channel(s)

This routine will scroll all the image channels indicated in the channel mask to the specified location in XValue and YValue. In addition, if a Zoom factor is present, it will zoom those channels to the specified factor.

Format: Ipi_Scroll(Unit, ChnMsk, XValue, YValue [, Zoom])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Indicates success or failure of the operation

Parameters: Unit
Usage: Unit to Operate On
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
I/O Channel of the DeAnza unit. Returned by Ipi_AttUnit.

ChanMask
Usage: Channel Mask indicating which to scroll
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
The channel mask containing a one in the bit position corresponding to the channel(s) to scroll. (A 1 in bit 0 will scroll channel 0.)

XValue
Usage: X Scroll position
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
This value indicates the X screen coordinate that will now become the left boundary of the screen.

YValue
Usage: Y Scroll position
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
This value indicates the Y screen coordinate that will now become the top boundary of the screen.

Zoom
Usage: Zoom factor
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
This value indicates the new zoom factor. Zoom factor of zero indicates 1:1, a factor of 1 indicates 1:2, etcetera.

Description:

Ipi_Scroll will scroll the channels specified in the channel masks. Realize that this scrolled position does NOT affect reading and writing of images through Ipi_GetPic or Ipi_PutPic other than how they appear on the screen. That is the image will be loaded into the normal position and will appear scrolled.

However, this WILL affect DVP operations. The DVP will take its input starting at the position indicated by the XValue and YValue. The output of the DVP is not affected by the scrolling. Output from the DVP is controlled by Ipi_DvpOrigin. However, caution should be taken when performing a DVP operation that takes its input and output from the same channel when the channel is scrolled. Because the DVP does not work sequentially, if the scroll region and the origin region overlap, unexpected results may occur.

Modification History

03-Feb-1987	SLS	1.0	All new code
	SLS	1.1	Converted to new driver scheme
05-Oct-1989	SLS	1.2	Converted to IPX scheme

SetPicOps - Define Picture Display Options

This routine is used to define the picture display options for a particular image. They determine how the image is displayed.

Format: Ipi_SetPicOps(PicOps [, Option] [, . . .])

Returns: Usage: Returned status of the Operation
 Type: Longword (Unsigned)
 Access: Write Only
 Mechanism: By Value
 Indicates success or failure of the operation.

Parameters: PicOps
 Usage: Picture Options Longword
 Type: Longword (Unsigned)
 Access: Modify
 Mechanism: By Reference
 A variable to contain the various picture options that are set by this routine and other PicOps routines.

 Option
 Usage: Picture Display Option [Optional]
 Type: Longword (Unsigned)
 Access: Read Only
 Mechanism: By Reference
 An option mask to control display of the image. There can be from zero to any number of these, separated by commas.

Description:

Ipi_SetPicOps is used to set the picture display options for a particular image. There are a number of options that may be used, listed below. There are other routines that can modify the picture display options, also listed below. You should call Ipi_SetPicOps before calling the other routines, as Ipi_SetPicOps initializes the PicOps to its default values. It is possible to "add" and "delete" options to and from the already existing options, as shown below. If you wish to reset everything, not just bit settings, but the size, offset, dimension, etcetera, back to their defaults, then call Ipi_SetPicOps with just the PicOps argument and it will reinitialize the PicOps.

Picture Display Options

- Mask -	- If Present -	- If Absent -
Ipi_M_Color	Image is three bands	Image is one band
Ipi_M_BandIL	Bands Band Interleaved	Bands Pixel Interleaved
Ipi_M_LineIL	Bands Line Interleaved	Bands Pixel Interleaved
Ipi_M_WordData	Data is 16 Bit	Data is 8 Bit
Ipi_M_XYSwitch	Y is Primary Axis	X is Primary Axis
Ipi_M_XInvert	X Axis is Inverted	X Axis is Normal
Ipi_M_YInvert	Y Axis is Inverted	Y Axis is Normal

Other Picture Option Routines

Ipi_SetPicBands	Sets the number of multi-spectral bands
Ipi_SetPicSize	Sets the X and Y display size for the image
Ipi_SetPicLocation	Sets the X and Y display location for the image
Ipi_SetPicDimension	Sets the X and Y dimension of the image file
Ipi_SetPicOffset	Sets the X and Y offset within the image file
Ipi_SetPicHeader	Defines any image header information

Condition Values Returned:

SS\$_Normal Normal successful completion
 Ipi__BadParCnt Bad parameter count
 any returned by Lib\$Get_Um

Modification History:

Date	Auth	Ver	Modification
02-Aug-1988	SLS	1.0	All new code

Authors: Stephen L. Schultz

Examples:

Setting Multiple Options, Displaying with Level 2 Calls

```
Call Ipi_SetPicOps( PicOps, Ipi_M_Color )            ! A Color Image
Call Ipi_SetPicSize( PicOps, 1024, 1024 )           ! Changes size to 1k
Call Ipi_ShowPic( Unit, 'Food_1K.Pix', PicOps ) ! Display the Picture
```

Setting Options in a Variable, Displaying with Level 1 Calls

```
Call Ipi_OpenFile( FileBlock, 'Word.Pix', 0 )       ! Open a file
Call Ipi_GetFileInfo( FileBlock, Size )            ! Get the File Size
If ( Size .eq. 1024 ) Then
  Type = Ipi_M_WordData                            ! Monochrome Word Data
Else If ( Size .eq. 3072 ) Then
  Type = Ipi_M_Color + Ipi_M_WordData            ! Color Word Data
End If
Call Ipi_SetPicOps( PicOps, Type )                ! Set the Options
Call Ipi_DiskPic( FileBlock, FilePtr, PicOps )    ! Attach to Disk
Call Ipi_PutPic( Unit, ChnMsk, %Val( FilePtr ), PicOps )
Call Ipi_CloseFile( FileBlock )
```

"Adding" and "Deleting" Options

Original Call:

```
Call Ipi_SetPicOps( PicOps, Ipi_M_Color )
```

Subsequent Call that "adds" an Option:

```
Call Ipi_SetPicOps( PicOps, %Val( PicOps ), Ipi_M_BandIL )
```

Subsequent Call that "deletes" an Option:

```
Call Ipi_SetPicOps( PicOps, %Val( PicOps ), Not( Ipi_M_BandIL ) )
```

By placing the %Val(PicOps) as the first argument, it will pass along all of its current options, in addition to any other indicated by the other arguments. This is essential if you wish to switch between different options, but keep everything else the same, such as image size, location, etcetera.

Ipi_ShowPic - Displays a Picture from Disk to Image Memory

This routine loads a picture from disk to image memory according to the settings in the PicOps block.

Format: Ipi_ShowPic(Unit, Filename, ChanMask, PicOps
[, Default] [, Resultant])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Indicates success or failure of the operation.

Arguments: Unit
Usage: Unit to Assign and Attach
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Reference
Specifies the Unit to attach the DUP to. Unit is returned by Ipi_AttUnit.

FileName
Usage: Name of the Picture File
Type: Character Array
Access: Read Only
Mechanism: By Descriptor
Contains the name of the file to load the picture from.
It is passed to the File-Spec parameter of Lib\$Find_File to locate the actual file.

ChannelMask
Usage: Mask of Channels
Type: Longword (Unsigned) [Array Length N]
Access: Read Only
Mechanism: By Reference
A channel mask is used to indicate which channels to load to. Each bit in the 32-bit mask corresponds to a channel on the Unit. If the bit is set, data will be sent to that channel. If this is a color or multiband image, ChnMsk must be an array with one element in the array for each band in the image (3 for a color image). If the user specified Ipi_M_WordData, that doubles the number of necessary ChnMsk elements, with each pair of elements representing the High and Low order bits of the single word of data.

PicOps
Usage: Picture Options Block
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
Specifies the address of the Picture Options Block that is defined by Ipi_SetPicOps and related calls.

Default
Usage: Default File Spec [Optional]
Type: Character Array
Access: Read Only
Mechanism: By Descriptor
Contains any default specifications for the file name.
It is passed to the Default-Spec parameter of Lib\$Find_File

to locate the actual file.

Resultant

Usage: Resultant File Spec [Optional]

Type: Character Array

Access: Modify

Mechanism: By Descriptor

Contains any related file specifications for the file name. After parsing the filename, this will contain the resulting file specification that was actually located by Lib\$Find_File. It is passed to the Related-Spec parameter of Lib\$Find_File to locate the actual file, and then the Result-Spec parameter from Lib\$Find_File is copied into it upon completion.

Description:

Ipi_ShowPic will load an image to the screen according to the settings in PicOps. A standard monochrome, 512x512 image would be loaded to standard image memory with the first line in the file being loaded to the top of the screen, down to the last line in the file being loaded to the bottom of the screen. The direction of loading may be modified by the following PicOps settings:

- Ipi_M_XYSwitch - Switches the X and Y axis in effect rotating the image about the line Y = -X
- Ipi_M_XInvert - Loads the X Axis in backwards
- Ipi_M_YInvert - Loads the Y Axis in backwards

The position on the screen and size of the image on the screen may be modified by other PicOps routines:

- Call Ipi_SetPicSize - Defines the X and Y Size of the image on the screen.
- Call Ipi_SetPicLocation - Defines the X and Y Location of the image on the screen.

*Note: Ipi_M_ExtMem must be set in the PicOps if you setting the Size or Location would cause the image to extend beyond the 512 x 512 standard memory.

The data window within the image file may be modified by the following PicOps routines:

- Call Ipi_SetPicDimension - Defines the X and Y Size of the image in the file. Only the portion corresponding to X and Y Size will be displayed.
- Call Ipi_SetPicOffset - Defines the X and Y Offset within the image file that the displayed portion will be removed from.

By default, the Size and Dimension of an image are the same, setting the size will automatically set the dimension, unless the dimension was explicitly changed. Also, by default, the Offsets are both zero, and the Location is the lower left corner of the display corresponding to 0 for the X and 0 for the Y.

Other settings that modify how the image is read:

- Ipi_M_WordData - Each pixel is represented, and loaded as 16-bits instead of 8-bits. This requires the user to specify twice the number of Channel Masks normally required for the image since now each mask must instead be a pair of masks, each pair consisting of the High and Low order bytes. So a Color Word image would require 6 channel masks.
- Ipi_M_Color - Each pixel is represented by three discrete elements, a Red, Green, and Blue element. There must be one channel mask for each element.

Call Ipi_SetPicBands - This can set the explicit number of bands for a multiband image. This should not be used in conjunction with the Ipi_M_Color option. There must be one channel mask for each band.

Condition Values Returned:

SS\$_Normal Normal successful completion
 Ipi__BadParCnt Bad parameter count
 Any of those returned by Ipi_Qiow

Modification History:

Date	Auth	Ver	Modification
04-Aug-1988	SLS	1.0	All new code
08-Aug-1990	SLS	1.1	Converted to IPX scheme

Authors: Stephen L. Schultz

Example:

Program ShowColor

Implicit None
 Include 'Ipi_IpiLib'

Integer*4 Unit, Status, ChanMask, PicOps, Check
 Character*40 Filename, Default

Check = Ipi_M_Exit + Ipi_M_NoText
 Default = '*.Pix'

Write (6,*) 'Enter the Filename of the Color Image:'
 Read (5,'(a40)',End=900) Filename

Write (6,*) 'Enter the Red, Green, and Blue Channel Masks:'
 Read (5,*,End=900) ChanMask(1), ChanMask(2), ChanMask(3)

Status = Ipi_AttUnit(Unit)
 Call Ipi_ErrorCheck(Status, 'Error Attaching Unit',,, Check)

Status = Ipi_SetPicOps(PicOps, Ipi_M_Color)
 Call Ipi_ErrorCheck(Status, 'Error Setting Options',,, Check)

Status = Ipi_ShowPic(Unit, Filename, ChanMask, PicOps, Default)
 Call Ipi_ErrorCheck(Status, 'Error Displaying Image',,, Check)

Status = Ipi_DetUnit(Unit)
 Call Ipi_ErrorCheck(Status, 'Error Detaching Unit',,, Check)

900 Continue
 End

Ipi_StatHst - Compute Statistics of a Histogram

This routine will calculate a number of statistics for a Histogram in addition to producing a Cumulative Histogram. Because of the large number of parameters, they are arranged in three groups:

First Pass - Low, High, Maximum, Total
Average - Mean, Mode, Median
Moments - StdDev, Skew, Kurtosis

If do not wish to produce the Cumulative Histogram, simply save the space with an extra comma. If you wish to produce any of the values within a group, you must include all of the commas for that group even if they occur AFTER the parameter. See the Examples below.

Format: Ipi_StatHst(Histogram, [Cumulative,]
[LowValue, HighValue, MaximumCount, TotalCount,]
[Mean, Median, Mode,]
[StdDev, Skew, Kurtosis])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Indicates success or failure of the operation.

Parameter: Histogram
Usage: Source Histogram
Type: Longword (Signed) [Length 256]
Access: Read Only
Mechanism: By Reference
This is the Histogram to produce statistics on. The Histogram is an array from 0 to 255 where each element in the array contains the number of occurrences of the Value corresponding to the element number.

Cumulative
Usage: Cumulative Histogram
Type: Longword (Signed) [Length 256]
Access: Write Only
Mechanism: By Reference
This histogram will be produced from the Source Histogram. A Cumulative Histogram is one where each element in the Histogram equals the sum of the count of all elements up to and including that element.

Low
Usage: Low Value in the Histogram
Type: Longword (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the lowest Value whose Histogram element is non-zero.

High
Usage: High Value in the Histogram
Type: Longword (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the highest Value whose Histogram element is non-zero.

Maximum

Usage: Maximum Count in the Histogram
Type: Longword (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the maximum count within the Histogram.
A side note: the maximum count occurs at the Mode.

Total
Usage: Total Count in the Histogram
Type: Longword (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the total count within the Histogram.

Mean
Usage: Mean Value in the Histogram
Type: F-Floating (Signed)
Access: Write Only
Mechanism: By Reference
This value will receive the Mean value within the Histogram.
Mean is defined by the sum of all the values divided by the total count.

Mode
Usage: Mode Value in the Histogram
Type: Longword (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the Mode value within the Histogram. Mode is defined as the value that occurs the most often.

Median
Usage: Median Value in the Histogram
Type: Longword (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the Median value within the Histogram.
Median is defined as the value that has an equal number of values below and above it.

StdDev
Usage: Standard Deviation of the Histogram
Type: F-Floating (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the Standard Deviation of the Histogram.

Skew
Usage: Skewness of the Histogram
Type: F-Floating (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the Skewness of the Histogram.
*** Not Implemented Yet.

Kurtosis
Usage: Kurtosis of the Histogram
Type: F-Floating (Signed)
Access: Write Only
Mechanism: By Reference
This will receive the Kurtosis of the Histogram.

*** Not Implemented Yet.

Description:

Ipi_StatHst can produce a wide variety of statistics for a Histogram. They are organized into three groups:

First Pass - Low, High, Maximum, Total

Average - Mean, Mode, Median

Moments - StdDev, Skew, Kurtosis

If do not wish to produce the Cumulative Histogram, simply save the space with an extra comma. If you wish to produce any of the values within a group, you must include all of the commas for that group even if they occur AFTER the parameter. Therefore, the ONLY valid number of parameters are:

2 : Just produce a Cumulative

6 : Produce Low Pass

9 : Produce Averages

12 : Produce Moments

As stated above, you do not need to supply all the values for a particular group, just the commas. The examples below help clarify this. It is recommended, for speed reasons, that if you do not need any of the results from latter groups, simply end the call at the end of the last group you use a value from.

Condition Values Returned:

Ipi__BadParCnt - Bad parameter count

Ipi__HstEmpty - Histogram did not contain any values

Any of those returned by Ipi_Qiow.

Modification History:

Date	Auth	Ver	Modification
12-Aug-1988	SLS	1.0	All new code

Authors: Stephen L. Schultz

Examples:

Integer*4	Unit, Status, Channel
Integer*4	Histogram(0 : 255), CumHist(0 : 255)
Integer*4	Low, High, Max, Total, Mode, Median
Real*4	Mean, StdDev, Skew, Kurtosis

*

***** Retrieve a Histogram

*

Write (6,*) 'Enter the Channel:'

Read (5,*,End=900) Channel

Status = Ipi_AttUnit(Unit)

Call Ipi_ErrorCheck(Status, 'Error Attaching Unit')

Status = Ipi_Histogram(Unit, Channel, Histogram)

Call Ipi_ErrorCheck(Status, 'Error Attaching Unit')

Status = Ipi_DetUnit(Unit)

Call Ipi_ErrorCheck(Status, 'Error Detaching Unit')

*

***** Now get some Statistics

*

* All of the following calls are valid, although the last one is
* quite useless. The first one produces just the Cumulative. The
* second one produces the Cumulative, the Low and High Value. The

```

*      third one produces the Low and High values and the Maximum and
*      Total counts. Etcetera, up to the last one which produces everything.
*      Again, notice how when at least one value from a group is necessary,
*      all of the commas for the group must be included. And naturally, if
*      you don't want any of the values from an early group, but do want
*      values from a latter group, you must include all of the first groups
*      commas.
*
*      Realize of course, that in a real program, you would use only one
*      of the following calls here, or some variation of one of the calls.
*
      Status = Ipi_StatHst( Histogram, Cumulative )
      Status = Ipi_StatHst( Histogram, Cumulative, Low, High,, )
      Status = Ipi_StatHst( Histogram, , Low, High, Max, Total )
      Status = Ipi_StatHst( Histogram, Cumulative, ,, Max,, Mean, Mode, )
      Status = Ipi_StatHst( Histogram, , ,,, Mean, Mode, Median )
      Status = Ipi_StatHst( Histogram, , ,,, Mean,,, StdDev,, )
      Status = Ipi_StatHst( Histogram, , ,,, ,,, StdDev, Skew, )
      Status = Ipi_StatHst( Histogram, Cumulative, Low, High, Max, Value,
+                               Mean, Mode, Median, StdDev, Skew, Kurtosis )
      Status = Ipi_StatHst( Histogram, , ,,, ,,, ,,, )
      Call Ipi_ErrorCheck( Status, 'Error Calculating Histogram Statistics' )

*
***** End of Program
*
900      Continue
      End

```

Ipi_ViewChan - View Channel Monochromatically

This Level Two routine allows the user to view a single image channel monochromatically, or to reset to normal color viewing.

Format: Ipi_ViewChan(Unit [, Channel])

Returns: Usage: Returned Status of Operation
Type: Longword (Unsigned)
Access: Write Only
Mechanism: By Value
Completion status of the operation

Arguments: Unit
Usage: Unit to Operate On
Type: Longword (Unsigned)
Access: Read Only
Mechanism: By Reference
The Unit channel to perform the operation on. This value is returned via a call to Ipi_AttnUnit.

Channel
Usage: Channel to View
Type: Word (Unsigned)
Access: Read Only
Mechanism: By Reference
Indicates the Channel Number the user wishes to view. If this argument is not present, Ipi_ViewChan will reset to color viewing.

Condition Values Returned:

SS\$_Normal Normal successful completion
Ipi_BadParCnt Bad parameter count
Any of those returned by the Ip8Qw Call.

Description:

Ipi_ViewChan modifies the VOC output gun assignments so that a single channel is assigned to multiple output guns. When reset, Channel 0 is assigned to the Red Gun, Channel 1 to the Green Gun, Channel 2 to the Blue Gun, and Channel 3 to the Graphics Interpreter.

Modification History

Date	Auth	Ver	Modification
29-Jan-1987	SLS	1.0	All new code
23-Feb-1987	SLS	1.1	Added an optional Second Format
28-Jul-1988	SLS	1.2	Moved second format to Ipi_VocAssign Reset now performed when no args given

Authors: Stephen L. Schultz

Examples:

Program ViewChan

Implicit None
Include 'Ipi_IpiLib'

Integer*4 Unit, Status, Length
Integer*2 Channel
Character*10 Input


```

Write (6,*) 'Enter Channel to View, or just hit return to reset:'
Read (5,'(q,a10)',End=900) Length, Input

Status = Ipi_AttUnit( Unit )
Call Ipi_ErrorCheck( Status, 'Error Attaching Unit' )

If ( Length .eq. 0 ) Then
    Status = Ipi_ViewChan( Unit )
    Call Ipi_ErrorCheck( Status, 'Error Resetting View' )
Else
    Read ( Input(1:Length), * ) Channel

    Status = Ipi_ViewChan( Unit, Channel )
    Call Ipi_ErrorCheck( Status, 'Error Viewing Channel' )
End If

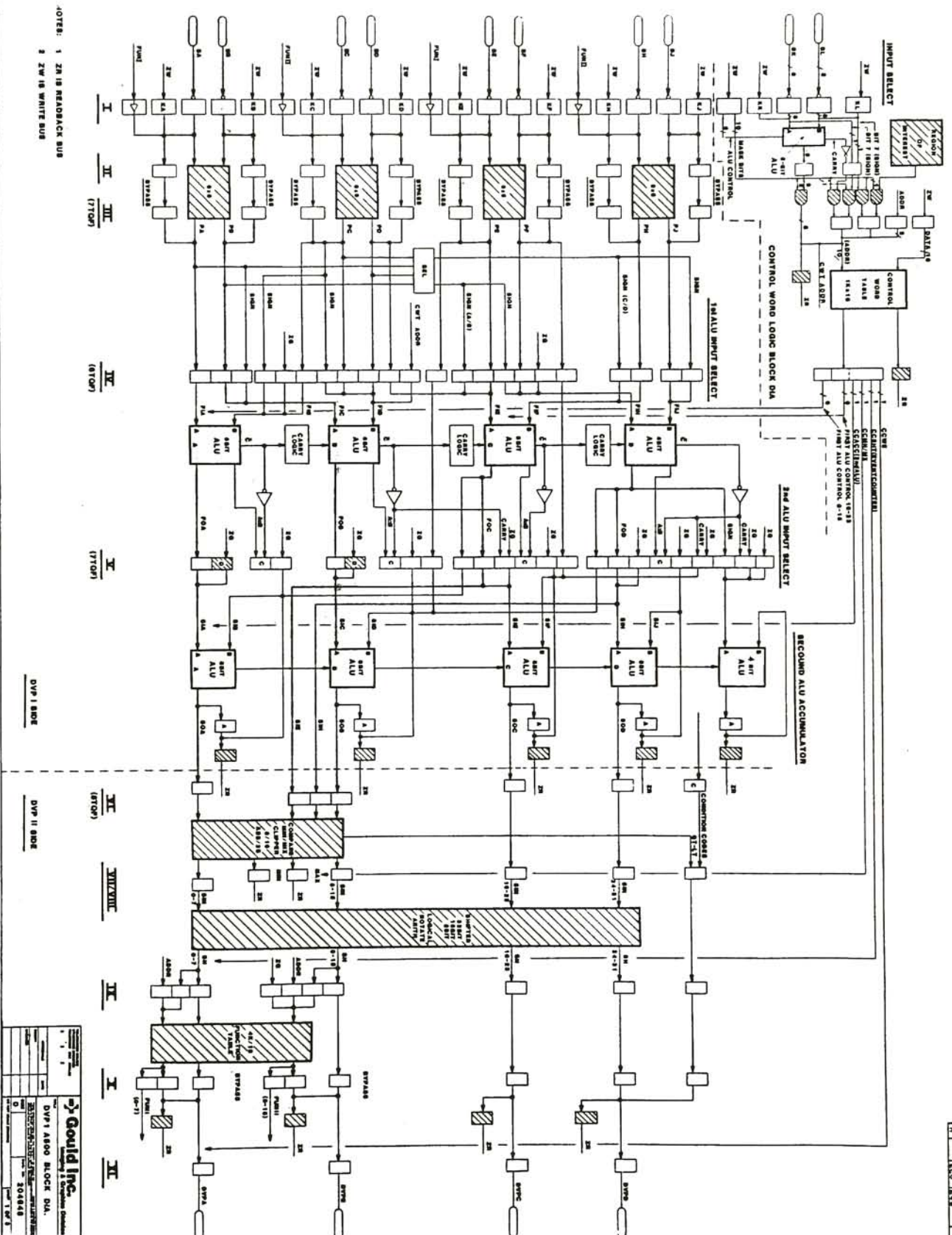
Status = Ipi_DetUnit( Unit )
Call Ipi_ErrorCheck( Status, 'Error Detaching Unit' )

End

```

Appendix E:

Block Diagram of DVP



GoULD INC. A Division of General Electric Company 1000 North 17th Street Philadelphia, PA 19104	
DVP A500 BLOCK DIA 304648	304648

FIGURE 15-3 DVP A500 BLOCK DIA

(Sheet 1 of 3)