1988

Binary image compression using run length encoding and multiple scanning techniques

Frank J. Merkl

Rochester Institute of Technology
School of Computer Science and Technology

Binary Image Compression Using Run Length Encoding and Multiple Scanning Techniques

by
Frank J. Merkl

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by: Peter G. Anderson
Professor Peter G. Anderson

Chris Comte
Professor Chris Comte

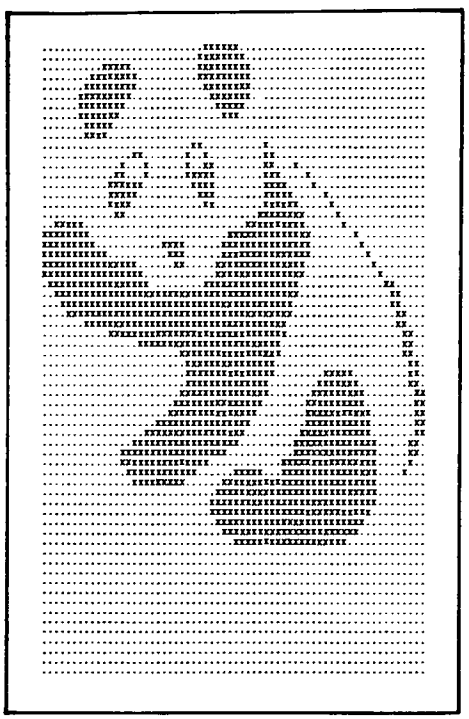
Rodger W. Baker
Professor Rodger W. Baker

May 16, 1988

Title of Thesis: Binary Image Compression Using Run Length Encoding And Multiple Scanning Techniques

I, Frank J. Merkl, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: May 19, 1988



T A B L E O F C O N T E N T S

Abstract

While run length encoding is a popular technique for binary image compression, a raster (line by line) scanning technique is almost always assumed and scant attention has been given to the possibilities of using other techniques to scan an image as it is encoded. This thesis looks at five different image scanning techniques and how their relationship to image features and scanning density (resolution) affects the overall compression that can be achieved with run length encoding. This thesis also compares the performance of run length encoding with an application of Huffman coding for binary image compression. To realize these goals a complete system of computer routines, the Image, Scanning and Compression (ISC) System has been developed and is now available for continued research in the area of binary image compression.

ACM Classification Codes:

I. 4 IMAGE PROCESSING

I. 4.2 Compression (Coding)

Chapter 1

Introduction

1.1 Problem Statement

This thesis explores the effectiveness of using different scanning techniques to compress a two-dimensional, binary graphics image using run length encoding. It examines the inter-relationships between run length encoding, scanning techniques, scanning densities and image features as they affect the overall image compression. It also contrasts run length encoding with an application of Huffman coding for binary image compression. In order to observe these relationships, to verify hypotheses, and to make general observations about binary image compression, a complete system of computer image processing routines, the Image Scanning and Compression (ISC) System, has been developed.

1.2 Background

The explosive growth in personal computers, computer networks and computer storage media, has prompted a strong interest in the processing of images as data, just as text and quantitative values have been processed as data since the inception of computers. One very crucial difference is the sometimes enormous amounts of data space required to represent the information content of a single image, such as

a paper document or a video screen, when it is viewed as a purely visual image, unrestricted as to content. These space requirements have a major impact on both the storage and transmission of images.

To process a visual image or picture it is first necessary to resolve it into individual picture elements called pixels (also called pels). Each pixel represents a small unit of image area. The image can then be scanned, pixel by pixel, with each pixel's color and light intensity represented by one or more bits of computer storage. The result is a digital picture of the original source image. For this thesis, pixels will only assume one of two values, 0 or 1, representing white or black, respectively. Consequently, each pixel value can be represented by one bit of computer storage. A digital picture of this type is called a binary picture, or simply a binary image.

An 8 1/2 X 11 inch document with a defined resolution of 200 pixels per inch requires half a megabyte of computer storage as a binary image; at 400 pixels per inch, two megabytes are required. To reduce these huge memory demands, encoding techniques are frequently employed to compress an image with either no loss or what is termed an "acceptable" loss in its information content. Corresponding to these two conditions are two major categories of data compression techniques: reversible and irreversible. Reversible techniques,

also called redundancy reduction techniques, remove redundancy from data in such a way that it can be later inserted and no information is lost [LYNCH]. Irreversible techniques, on the other hand, achieve much greater compression at the cost of an acceptable, but irrecoverable, loss in the average information content of the data; its "entropy" is reduced and, consequently, irreversible techniques are frequently referred to as entropy reduction techniques.

The entropy of an image may be viewed as the average information content of "messages" obtained by partitioning the image into disjoint sets of pixels. When the compression of an image precludes its exact restoration, the average information content of the messages that can be obtained from the image after it has been decompressed is less than before it was compressed. This thesis will only be concerned with reversible compression techniques and so the image entropy will be preserved in the compression process.

One of the simplest of all reversible data compression techniques is called run length encoding. In run length encoding repeated items in a data stream are replaced by a count of the items and, usually, one instance of the item itself. The longer the runs, the more effective the compression. Run length encoding is particularly effective for image compression since many images are dominated by a background color.

The simplicity of run length encoding stems primarily from the fact that it need not assume any image structure or inter-relationship between image pixels as do many of the more sophisticated encoding schemes. Although all image pixels must be scanned once, run length encoding imposes no restriction on the sequence in which they are scanned. Consequently, the scanning and encoding processes can be relatively independent and the selection of the scanning technique for any given image is virtually unrestricted.

Two important application areas for the compression of binary images that use run length encoding are facsimile transmission and graphics image storage and retrieval systems. For facsimile, any reduction in data volume due to compression easily translates into cost savings in reduced transmission time and/or channel bandwidth.

Chapter 2

Related Work

The simplicity and intuitiveness of run length encoding makes its origin difficult to trace to any one individual. Several variants exist but a scheme described by G. Held [HELD] that combines both run length values and unencoded source data in the same output stream appears to be the most common. In this scheme a special, reserved character or symbol in the output stream is used to indicate that the next two data items (usually bytes) are a source character followed by a run value indicating the number of consecutive times that source character occurs, a total of three bytes in all. Encoding would not be worthwhile, and would be precluded (i.e., the special character would not appear), unless the number of repeated occurrences of the source character exceeds 3. For example, if the special character is "#" then the character sequence AAAAABBCCCCCCCCC would be encoded as #A5BB#C9, a 50% reduction in size. This type of scheme is essential when the average run size in the input data stream is relatively small as, for example, with text where the only runs usually worth encoding are those made up of space characters. For binary graphics images, where large runs of one color or the other usually dominate, this scheme has several disadvantages: (1) a designated special character is required; (2) a run requires three bytes to represent it; and

(3) encoding and decoding routines are relatively complex.

A two-dimensional run length encoding scheme for general image compression (not necessarily binary images) has been developed by B. Pracht and K. Bowyer [PRACHT]. This scheme (actually two separate schemes) encodes either largest overlapping, or largest non-overlapping, rectangular blocks of pixels of the same color by their corner coordinates. Both schemes, however, rely on some way of marking pixels as either processed or unprocessed as the image is encoded. Since pixels in binary images are represented by bits, which can only assume two values, there is no way to mark a pixel as having already been scanned without introducing a secondary array.

A scheme for compressing binary images which does not use run length encoding has been pursued by E. Davies and A. Plummer and is based on still earlier work by Rosenfeld and Pflatz [DAVIES]. In this approach a binary image is reduced to a minimal set of non-background points whose integer values are equal to their distance from the nearest background point. This provides high compression for certain types of images (a circle could be reduced to a single point with a value equal to the radius in pixels) but the encoding and decoding algorithms are quite complex in comparison to run length encoding.

Still another approach to binary image compression, presented by M. Kunt and O. Johnsen, is known as block coding [KUNT]. In this scheme, fixed size, rectangular blocks of pixels are encoded using variable length code words which are based on the frequency of occurrence of each block configuration. For blocks larger than 3 x 3 pixels (512 possibilities), optimum Huffman coding soon becomes impractical, so a sub-optimum coding scheme is introduced. Since, in most images, the solid background block occurs most often, in the sub-optimum coding scheme this block configuration is simply assigned the code word 0. All other configurations are assigned fixed length code words beginning with 1 followed by the block configuration itself represented as a bit configuration. For example, a 3 x 3 pixel block in a "plus sign" configuration would be represented by the code word 1010111010. Sub-optimum block coding schemes are simple to implement and have proved to be very efficient with average code word lengths approaching (minimally) those generated by optimum Huffman coding [KUNT].

The International Telegraph and Telephone Consultative Committee (CCITT) has proposed a standard for one-dimensional compression of binary images for facsimile transmission which specifically recommends the use of run length encoding [CCITT]. Instead of transmitting run values directly, the CCITT standard goes a step further, specifying that the run values themselves be encoded using a modified Huffman coding

technique with fixed code words. Tables of the fixed code words are included in the standard. A modified Huffman coding technique is required since it would be impractical to try to generate Huffman code words for all of the possible run values that could occur.

The CCITT has also proposed a standard for two-dimensional, binary image compression schemes which roughly specifies that only the positions of pixels that have changed color from the previously scanned line be transmitted. This and other two-dimensional encoding schemes exploit the fact that in most images, with a reasonably high resolution (scanning density), few pixels change color from line to line. Both the one- and two-dimensional compression schemes proposed in the CCITT standard assume a line by line raster scanning technique. Both schemes are expected to eventually be the recognized standards for facsimile transceivers.

Chapter 3

Image Compression

3.1 Digital/Binary Images

Image compression techniques work with digital images which have usually been created from original source images, such as paper documents or photographs. They are created either prior to, or simultaneously with, the compression process itself. A digital image is simply a collection of quantized values representing the color and light intensity of the original image at points (pixels) selected or scanned in a grid-like fashion. The quantization process converts the continuous pixel magnitudes into a fixed, finite set of magnitude levels each of which can be represented digitally by one or more bits of computer storage. A digital image could also represent a video terminal screen image in which case quantization would normally not be necessary.

A certain amount of information is almost always lost from the original image as a result of the quantization process. Some pixels will be defined on units of area of the original image that are not totally of one color (i.e., on edges) forcing the image scanner to make a decision which will always incorrectly represent the area to some degree.

Pixels of digital images that exhibit a gray scale require multiple bits to represent the various shades of color (brightness) from solid white to solid black. For example, an image with 8 defined levels of brightness requires 3 bits to represent the value of each pixel. In contrast, the pixels of a binary digital image only assume two color values, black or white, and hence each pixel's value can be represented by a single bit of computer storage. An image represented in this way is said to be bit-mapped. All of the digital images considered by this thesis are binary images.

To create a binary digital image from an original source image, such as a paper document, the document is scanned at a density measured in pixels per inch, by a scanner that measures the intensity of reflected light. The analog output of the scanner is put through a bilevel quantization process which uses a fixed threshold value to classify the scanner signal as black (level 1) when it is less than or equal to the threshold, and white (level 0), otherwise. Specifically, the analog scanner output f(x,y) is quantized into the bilevel signal b(x,y) according to the following rule:

$$b(x,y) = 1 \quad \text{if } f(x,y) <= t$$
$$= 0 \quad \text{otherwise}$$

where t is the threshold level and x and y are the pixel's coordinates [TING]. A dark pixel with little light reflectance would have a low value for f(x,y) and would be

represented by a 1. The selection of the "best" threshold value, t, for any given image is of critical importance: too low and image detail is lost, too high and false detail is introduced.

For a given image the information loss from bilevel quantization is greater than that from gray scale quantization and usually results in some image distortion. Two methods used to minimize this distortion are: (1) majority logic smoothing, where the "smoothed" value of a pixel is set to black when more than half of the pixels in a neighborhood defined around it are black; and (2) logical smoothing, where a pixel's smoothed value is determined by the presence or absence of certain configurations of the black and white pixels in a neighborhood around it [TING].

For this thesis, it is assumed that images to be scanned and compressed already reside in a bit-mapped storage area. Consequently, any scanning process required to resolve and quantize an original source image into binary valued pixels (to create the binary image) is considered to be a pre-scan. A pre-scan would be required for real world original images, such as paper documents or photographs, but would not be necessary if the original image were a monochrome video terminal screen. In any case, the term "original image" will henceforth refer to a stored bit-mapped binary image.

3.1.1 Image Characteristics

Specifically, for this thesis the original images considered for scanning and encoding are restricted to having the following characteristics:

(1) Images can be represented by two-dimensional grids of equally spaced pixels.

(2) Image colors are black and white only (no gray scale).

(3) Images are of a graphics nature, made up of regions of one or more pixels of the same color and unrestricted as to shape.

(4) Images are square with side length equal to a power of two. This restriction is simply to maintain a proper balance between theory and programming and is really only a requirement for two of the five scanning techniques to be presented.

Graphics images are made up simply of regions of pixels, no symbols are defined. Typical graphics images are charts, graphs, posters, maps, drawings, etc. Text may also be included but only when the text is not expected to be interpreted as individual characters.

An important advantage in problem simplification results from the restriction to two image colors: in the compressed representation of an image, alternate run values can represent alternate colors and there is no need to store an instance of the color itself. There is, of course, a requirement to identify which color is represented by the first run value. This simplification does not lessen the essence of the problem and an extension to any number of colors can be achieved by simply following each run value with a value indicating the particular color the run represents. Also, although not pertinent to this thesis, by storing only run values, the polarity of the entire image can be reversed upon decompression by simply changing the defined color of the first run value.

No attempt has been made in this thesis to standardize on an 8 1/2 x 11 inch or a similar sized square document. Instead, the test images are perceived to be either small (square) windows into full-size documents or video screens, or full-size images in their own right. The perception of windows permits the simulation of higher scan densities than could reasonably obtained without an actual scanner. For example, a test image labeled "text" (see Appendix A-9) consists of three script letters in a 64 x 64 pixel image area. In a standard typeset page these three letters would take up about 3/16 of an inch, so the test image simulates a small window into the page at a scanning density of approximately

340 (i.e., 64 x 16/3) pixels per inch.

Although the highest input scanning density/resolution for the test images used in this thesis was 64 x 64 pixels (4,096 pixels), each test image was scaled up to 128 x 128 pixels (16,384 pixels) for one series of compression runs. The 64 x 64 input scanning density was not a limitation of the ISC system, but a practical limit for manually designing and keyboarding the test images in the absence of a real front-end scanner. By way of comparison, a standard television screen has 525 lines of 512 pixels each and the new IBM graphics standard, VGA, displays 640 x 480 pixels (in 16 colors) on a monitor screen. The CCITT recommended resolution standard is 1728 pixels per 215 millimeter scan line (approximately 200 pixels/inch) horizontally and 3.85 scan lines per millimeter (approximately 100 pixels/inch) vertically.

3.1.2 Image Generation

The ISC system uses test images from three sources:

(1) Directly from the input terminal keyboard (simulating a real time scanning device).

(2) Using the contents of a stored image file, initially created by keyboard, scanner or other means.

(3) Selected from a set of internally generated test images of basic shapes.

By means of (1) or (2), virtually any image could be made available to the scanning and encoding routines.

An ISC system scaling routine automatically scales an input or generated image to a preselected system image size, specified in pixels (see Section 4.3).

3.2 Scanning Techniques

As previously mentioned, scanning for pixel quantization and scanning for compression are viewed as separate processes. Since this thesis is concerned only with the latter, it will always be assumed that scanning for quantization has preceded or is being performed simultaneously (and transparently) with scanning for compression. Consequently, henceforth, whenever the term scanning is used, it will mean scanning for compression.

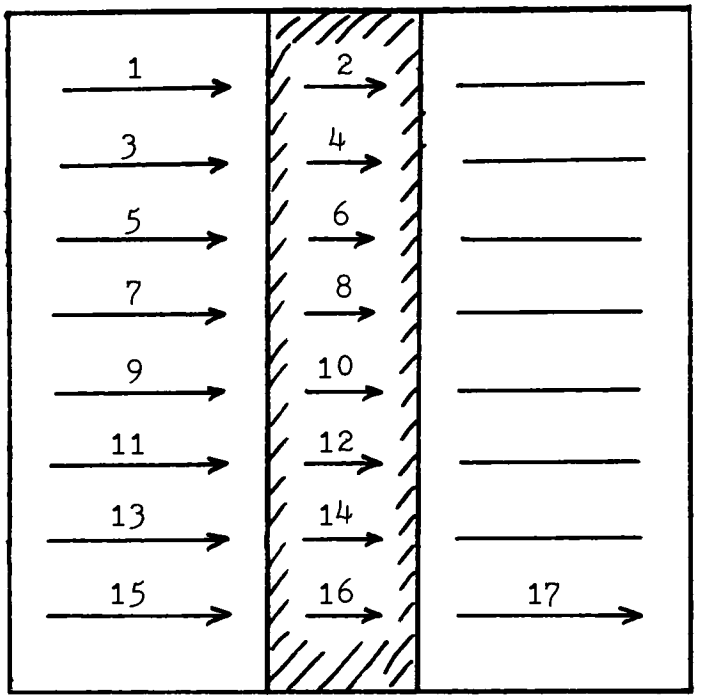
The most common technique for scanning images is from left to right, top to bottom, and is called raster or horizontal scanning. However, if an image is being scanned for compression using run length encoding, there need not be any restriction on the sequence in which the image pixels are actually scanned. In general, for any given image, different scanning techniques will result in different degrees of

compression. For example, the number of runs (consecutive pixels of the same color) encountered by raster scanning an image consisting of a single vertical line (any thickness) is equal to two times the number of scan lines, plus one. Simply changing to vertical scanning, top to bottom, left to right, reduces the number of runs to three, regardless of the number of scan lines (see Figure 1). Besides horizontal and vertical scanning, other scanning techniques can be used with run length encoding, each achieving a different degree of compression, as will be shown.

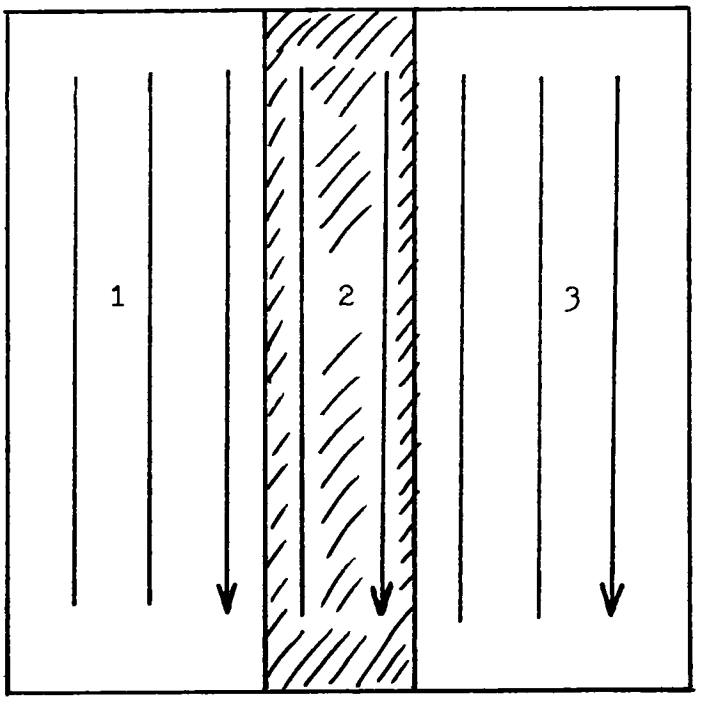
A set of five scanning techniques have been implemented in the ISC system. Each technique follows its own unique path in scanning an image, pixel by pixel.

3.2.1 Horizontal Scanning

Horizontal image scanning, usually called raster scanning, requires that an image be scanned, pixel by pixel, from left to right by position, and from top to bottom by line (see Figure 2). A variant is to scan alternate lines from left to right and then, on the return, from right to left. This would probably provide a slightly better compression for run length encoding (since the scanner would always be scanning adjacent pixels) but has not been explored so as to retain a link to the much more prevalent raster scanning.

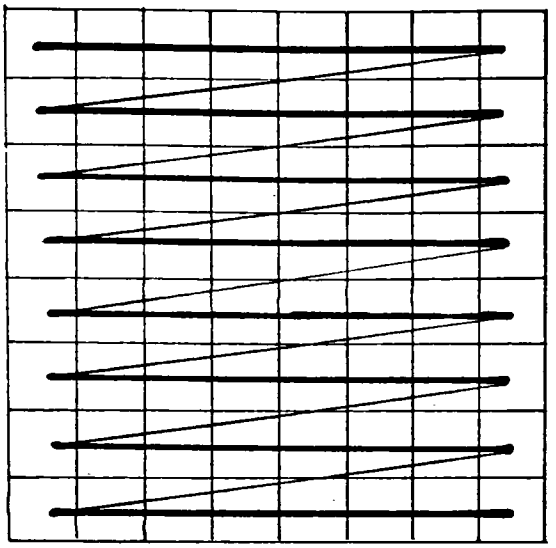


Horizontal Scanning



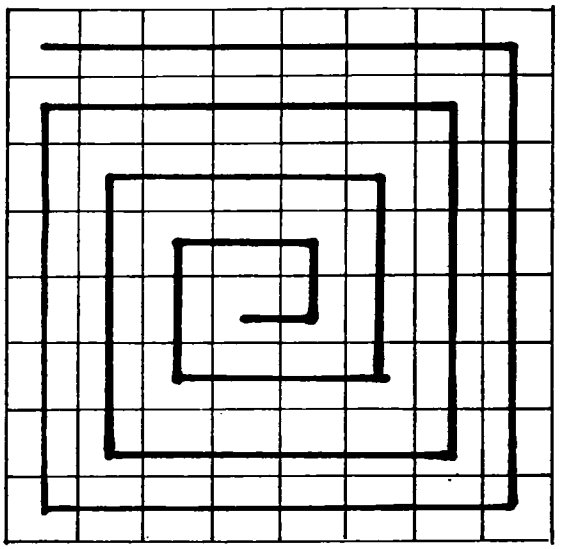
Vertical Scanning

Figure 1

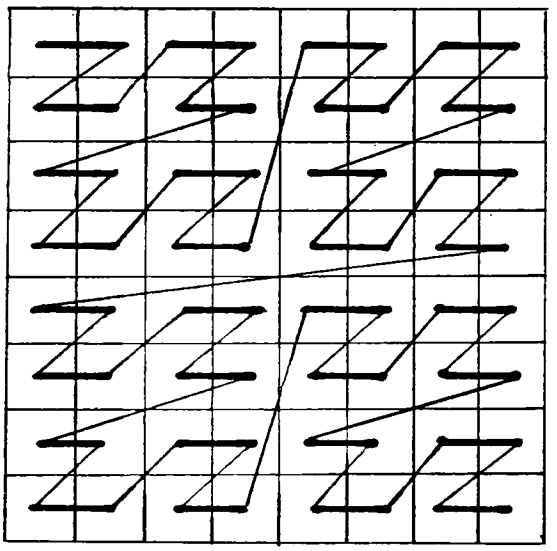


Horizontal Scanning

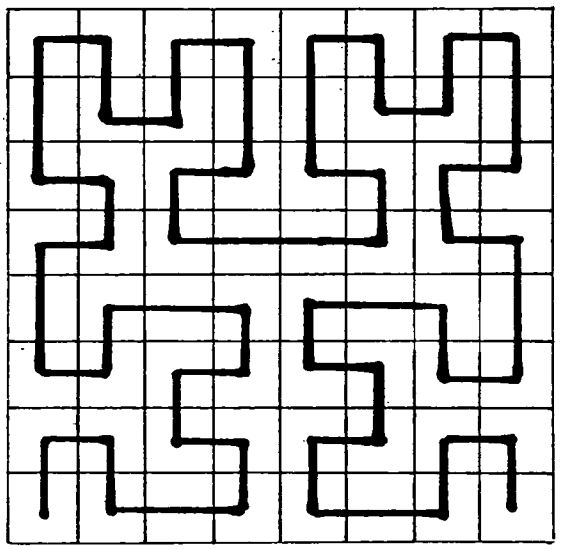
(Vertical Scanning at 90°)



Perimeter Scanning



Quadrant Scanning



Hilbert Scanning

Image Scanning Techniques at 8 X 8 Pixels

Figure 2

3.2.2 Vertical Scanning

Vertical scanning requires that an image be scanned from top to bottom by line, and from left to right by position. Vertical scanning can be viewed as simply a variant of horizontal scanning with the orientation of either the image or the scanner shifted by 90 degrees. This slight change in orientation, however, can result in dramatic differences in the compression achieved for certain images.

3.2.3 Perimeter Scanning

Perimeter scanning is based on scanning the perimeter of an image, edge by edge, continually shrinking in until there are no longer any pixels left to scan. Edges are scanned so that consecutive pixels are always adjacent: top (left to right), right (top to bottom), bottom (right to left) and left (bottom to top). As each edge is scanned it is effectively chopped off, eventually reducing the image to nothing (see Figure 2). Perimeter scanning can be very effective since most real world images, paper and video screens, have some kind of borders made up of background pixels. Perimeter scanning is also not constrained to square images.

3.2.4 Quadrant Scanning

Quadrant scanning involves scanning an image, which is necessarily a square with side length equal to a power of 2,

in progressively larger and larger quadrants (see Figure 2). It is related to the construction of image data structures called quadtrees which are created in the reverse manner: by recursively decomposing a square image into smaller and smaller quadrants until each quadrant contains only pixels of a single color [SAMET]. Quadrant scanning, as presented in this thesis, uses a technique based on Morton sequencing [LAUZON].

Morton sequencing was first introduced in 1966 by G. M. Morton of IBM and is a cell numbering system which attempts to place cells that are near to each other in two-dimensional space, close together in one-dimensional space. It assigns consecutive numbers to cells, beginning with zero, such that, in the number's binary representation, the odd numbered bits denote the cell's coordinate along one axis and the even numbered bits along the other. So, given a cell's assigned number, its position can be obtained, and vice-versa. For example, assuming a coordinate system with the origin (0,0) in the upper left hand corner, the cell at position (5,3) = (0101,0011) would be assigned the Morton sequence number 27 = 00011011.

Morton sequencing can be used to effect quadrant scanning. If the numbers from zero to the total number of image pixels, minus one, are considered to be Morton sequence numbers, and are processed consecutively with the odd and

even numbered bits (i.e., bits are numbered b7,...,b0) designated to represent image line and position, respectively, then the scan path through the image will be exactly the same as if the image were being covered by larger and larger quadrants.

3.2.5 Hilbert Curve Scanning

A Hilbert curve, also called a Peano curve or a continuous space filling curve, is a curve whose discovery is credited to mathematician David Hilbert. A Hilbert curve traces an unbroken path through a square matrix of cells passing through each cell only once and always passing to an adjacent cell (top, bottom, left or right), but not to a diagonal cell (see Figure 2). It differs from many other unbroken path curves in that it exhibits a strong sense of regionalism, totally exhausting a region in both dimensions as it spreads across the entire image. After hilbert scanning has scanned the first M pixels of an image, it will have totally exhausted a region with a diameter equal to the square root of M. As with quadrant scanning, Hilbert curve scanning requires that an image be square and have a side length equal to a power of 2.

Because horizontal and vertical scanning operate on only one line (or column) at a time they are classified as one-dimensional scanning techniques. Quadrant and hilbert scanning would then be classified as two-dimensional. Perimeter

scanning is something of a hybrid and cannot really be classified as either.

3.3 Encoding Techniques

The goal of image compression is to represent the information content of an image in less space (as measured in bits) then the image requires in its natural form (as measured in bits representing pixel values). Encoding is the process that brings about this transformation. The original image is then restored from its encoded representation prior to display. In this thesis two types of encoding are explored: run length encoding and Huffman (en)coding. The principle area of interest is run length encoding with Huffman coding providing a basis for comparison.

The effectiveness of image compression is expressed as a ratio relating the number of bits needed to store an image before compression and the number of bits needed after it has been scanned and encoded. Specifically, the compression ratio (C. R.) is defined as:

C. R. = bits in original image / bits in compressed image

The higher the compression ratio the better the compression. A compression ratio of 3 means that, on the average, 3 pixels in the original image are represented by 1 bit in the compressed image. The ISC system calculates and displays the compression ratio as well as other statistics for each image

compressed.

3.3.1 Run Length Encoding

In the ISC system, as an image is scanned it is simultaneously run length encoded, producing a sequence of integer run values which are then stored as an integer array. Each run value represents the length of a run of either black or white pixels and alternate run values represent alternate colors.

Although not a restriction of the ISC system, for this thesis, the default storage allocation for each run value is an 8-bit byte with multi-byte constructions for run values larger than 255. Consequently, run length encoding an image would not be of much value unless the average run value is greater than 8. Rather than mix run values and unencoded runs of pixels in the compressed image representation (see Chapter 2), the run length encoding routine of the ISC system has been designed to terminate encoding, and to store the entire image unencoded, whenever the point is reached where the storage space for the run values will exceed the storage space for the image uncompressed. It is recognized that this could cause a problem if an image, as it is being encoded, is simultaneously being transmitted, rather than just being stored, and special retransmit procedures would have to be in effect.

Since in the compressed representation of an image, alternate run values represent alternate colors (no instance of the color itself is stored), a problem arises on how to indicate which color is represented by the first run value. The solution proposed in the CCITT standard is to always assume the first run value represents the color white and to set it equal to zero if the first color happens to be black. In the ISC system, however, the number zero is already being used for multi-byte run values (see Section 4.2).

Another problem is that, although the ISC system in its present configuration compresses an image using all five scanning techniques/routines, in practical applications only one, but any one, might be used. Consequently, the identity of the scanning technique that has been used to scan an image must be somehow preserved with the image's compressed representation so it can be properly decompressed. Also, as was previously mentioned, an image may have no compression at all and, if so, this too must be indicated.

To satisfy the above requirements, the first element of each compressed image's run values array has been designated to indicate both the scanning technique used, and the color of the first run value (which is now the second array element). The specific values the first array element may hold are:

Value	Scanning Technique	First Color
0	Unencoded	Either
1	Horizontal	White
2	Horizontal	Black
3	Vertical	White
4	Vertical	Black
5	Perimeter	White
6	Perimeter	Black
7	Quadrant	White
8	Quadrant	Black
9	Hilbert	White
10	Hilbert	Black

3.3.2 Huffman Coding

Huffman coding is a technique for data compression introduced by D. A. Huffman in 1952. In Huffman coding, fixed size data elements (e.g., symbols, characters, numeric values, etc.) are encoded by assigning code words that vary in size according to each data element's frequency or probability of occurrence. Data elements that occur frequently are assigned short code words while those that occur infrequently are assigned long code words. Huffman's technique optimizes the selection of the code words so that the stream of encoded data elements will be of minimum length. For any given set of data elements and corresponding frequency distribution, the Huffman code words generated are not unique

but the compression achieved is. A key feature of Huffman coding is that no code word is the beginning of a longer code word which means a coded data stream can be instantaneously decoded.

If the data elements are combined and Huffman coded in blocks, the average number of bits to encode each data element is reduced and approaches a lower bound known as the entropy of the information source (with the blocks then considered as messages from the source). The lower bound is attained only if the probabilities of occurrence for the blocks are all negative powers of 2 and, of course, sum to 1 (e.g., 1/2, 1/4, 1/8 and 1/8). This is not normally the case.

The entropy of an information source is calculated as follows [CORBIN]:

> If the probability of occurrence of a data element is p, then the number of bits b to encode it is given by:
>
> $$b = f(-\log_2 p)$$
>
> where f(x) is the closest integer >= x. (This result is due to early work in information theory by C. Shannon at AT&T.)
>
> For example, in English text the probability of the letter "e" is 0.13, so

$$b = f(-\log_2 0.13) = f(2.94) = 3 \text{ bits}$$

The entropy, H, which is the average number of bits to encode each data element, is then:

$$H = -\text{Summation}\ [\ p(i) * \log_2 p(i)\ ] \text{ bits}$$

where p(i) is the probability of occurrence of the i-th data element and the summation is over all data elements. The entropy for the English alphabet and the space character in general text is about 4.1 bits.

Although blocking reduces the average number of bits to encode each data element, it exponentially increases the number of possible code words, quickly resulting in both large increases in processing time and a very large code word lookup table.

For image compression, Huffman coding can be used to directly encode small fixed size blocks of pixels, as is done in this thesis, or it can be used in conjunction with run length encoding to encode the run values, as is specified in the CCITT recommended standard [CCITT]. Pixel blocks can be directly encoded using the standard Huffman algorithm [LYNCH] based on the frequency of occurrence of each block configuration. The optimum code words so derived can then be used to encode the image, usually by table lookup. These code words, however, are unique to the image and a new set of (optimum)

code words would have to be generated for the next image. More importantly, the code words must be preserved and stored and/or transmitted with the image in order to be able to later decode it.

An alternate approach is to first establish a fixed set of code words and then use it to encode all subsequent images. Fixed code words are independent of the image being encoded, significantly reducing the encoding time and eliminating the need to store or transmit a separate code word table with each compressed image. One way to create a "good" set of fixed code words is to base it on the total frequency distribution of a group of images of the same type as the target group. Obviously, compression using fixed code words would generally not be optimum. The CCITT recommended standard uses fixed code word tables based on the results of encoding a group of 11 "typical" documents [CCITT].

Blocks of four contiguous horizontal pixels, a total of 16 possible configurations, were coded using Huffman code words. Both fixed and optimum Huffman code words were used to encode all test images, not only to provide a basis for comparison to run length encoding, but also to compare to each other. The fixed code words were derived from the combined frequency distribution of 4-pixel blocks contained in three 64 x 64 base resolution test images called panda, text

and flowchart (see Appendix A-8, A-9 and A-10).

3.4 Decoding Techniques

Encoding data for compression would be of little use without a corresponding decoding process. Consequently, a complimentary image decompression/decoding routine for run length encoding is included in the ISC system. Only one decompression routine is required since it uses the same scanning routines that are used to compress an image, but reverses the process, converting an array of run values into a pixel/bit integer array to reconstruct an original image.

To verify the integrity of the scanning and encoding routines of the ISC system, each decompressed image is compared to the original uncompressed image to ensure an exact replica has been produced.

Chapter 4

ISC System Architectural Design

The ISC system performs the basic function of compressing a binary image by the application of scanning and encoding techniques. In its present configuration the ISC system serves primarily as a research tool; every image is compressed using run length encoding and all five of the system's scanning techniques. In addition, every image is compressed using Huffman code words.

4.1 System Configuration

The principle components of the ISC system (see Figures 3a and 3b) are:

(1) A main (user interface/driver) program.

(2) A set of five scanning routines.

(3) Three image generation methods.

(4) Run length encoding and decoding routines.

(5) A Huffman coding routine.

(6) Auxiliary routines.

Each of the five scanning routines uses a distinct technique to scan a bit-mapped, two-dimensional, binary image

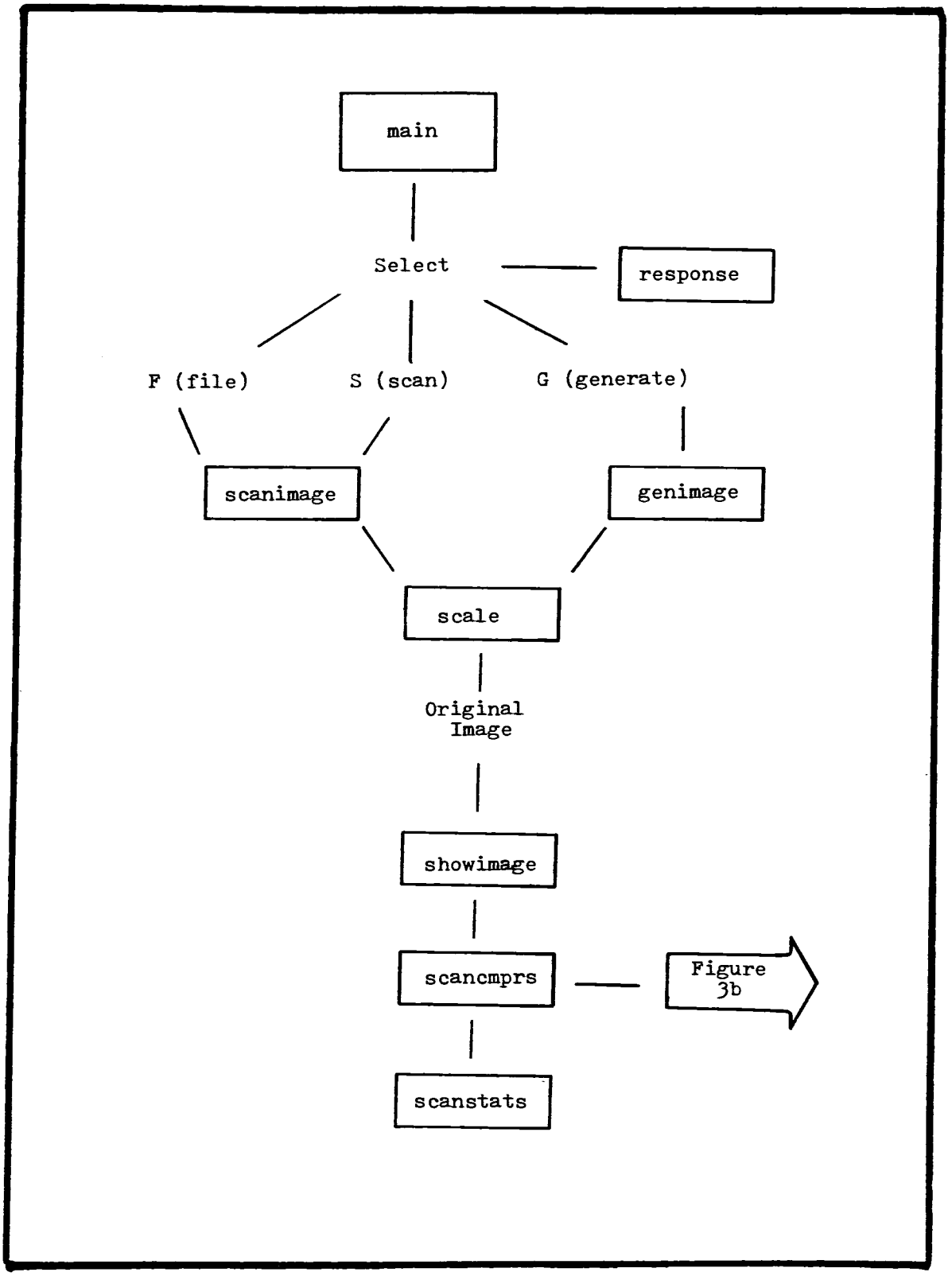

main
Select
response
F (file)
S (scan)
G (generate)
scanimage
genimage
scale
Original
Image
showimage
scancmprs
Figure
3b
scanstats

Figure 3a

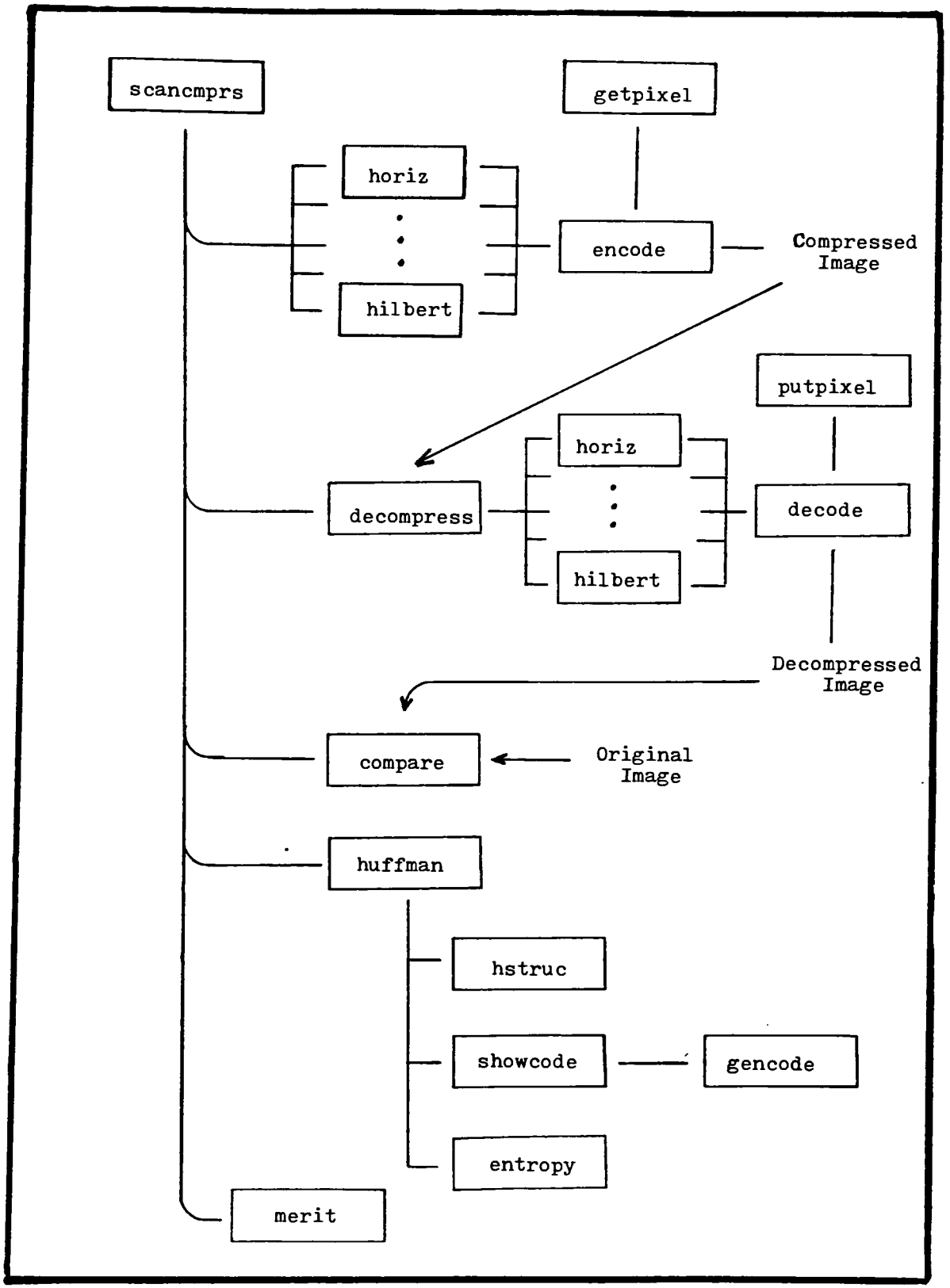

scancmprs
getpixel
horiz
encode
Compressed Image
hilbert
putpixel
horiz
decompress
decode
hilbert
Decompressed Image
compare
Original Image
huffman
hstruc
showcode
gencode
entropy
merit

Figure 3b

provided by means of a terminal keyboard, a stored file, or an internal ISC system image generation routine. The scanning routines provide a steady stream of pixel coordinates to the run length encoding routine which then produces a compressed representation of the image as an integer array of run values. The original binary image is regenerated (as a bit-mapped integer array) whenever its compressed run values array is passed to the run length decoding routine.

The Huffman coding routine compresses an image by encoding 4-pixel blocks using both optimum (unique to the image) code words and fixed code words. The routine does not actually produce a compressed representation of the binary image, instead it generates both fixed and optimum sets of code words, and then calculates, if the image were to be compressed, the size of the compressed image in bits, the compression ratio and the compression efficiency, using both sets of code words. The entropy of the image (as an information source) is also calculated.

The separation of the scanning, encoding and image generation routines allows the ISC system to be open-ended. Additional scanning routines and encoding routines can added to the system without difficulty, These can then investigated for their own merits or in comparison with the existing rou-

tines, using virtually any desired image.

4.2 Design Implementation Features

The primary data structure of the ISC system is the integer array. Although not a limit of the host computer, nor the ISC system, an 8-bit byte was selected as the (simulated) storage unit for all image arrays: original, compressed and decompressed. Compressed image run values that exceed the limit of an 8-bit byte are stored using a multi-byte construction in the following manner: since the value zero cannot logically occur as a valid run value it is used as an escape character, serving to indicate that the next two bytes are required to compute the actual run value; the first of these two bytes contains the highest multiple of 255 less than the run value; the second contains the remainder after dividing the run value by 255.

Each compressed image array has two overhead elements: the first array element indicates both the scanning method used and the color represented by the first run value; the last array element is the array terminator. The first is considered in calculating the compression ratio, the last is not since it may be implemented in any number of ways on different systems.

While all images, prior to scanning, are perceived as two-dimensional bit arrays, they actually reside in one-

dimensional 8-bit integer arrays. The two forms are equivalent in that the integer arrays are read and written on a bit by bit (i.e., pixel by pixel) basis. The bit value 1 has arbitrarily been chosen to represent the color black and the bit value 0 the color white; the color white is also assumed to be the background color.

4.3 Configuration and Control Options

Several parameters (symbolic constants) have been defined to provide flexibility in changing the ISC system's run time "dimensions." These are:

(1) HPIX and VPIX: set the vertical and horizontal dimensions in pixels of the original image. VPIX must be equal to HPIX and both must be a power of 2.

(2) SCANSIZE: sets the size in pixels of a horizontal scan line of the original input image. If SCANSIZE is less than HPIX the scanned input image is automatically scaled up to HPIX x VPIX.

(3) SHOWSIZE: sets the display size in pixels of the original image when the function showimage is invoked. The maximum SHOWSIZE is 64 x 64; if the original image size (as set by HPIX and VPIX) is greater than 64 x 64 then the display

is scaled down to fit.

(4) ISIZE: sets the maximum size of the system image array elements in bits.

Two additional system control variables whose values are set by the user as the result of system prompts are showscans and showarrays. A response of "yes" to showscans will cause pixel values to be displayed as an image is being scanned. A "yes" response to showarrays causes the integer arrays representing the original, compressed and decompressed images to be displayed. While these variables are extremely useful for system development and research purposes, their use is impractical for high scanning densities.

A typical display for a terminal session using the ISC system to compress a binary image may be seen in Figure 4.

```
                IMAGE SCANNING AND COMPRESSION

Select the means of providing the target image:

       S = Scanner or keyboard

       F - Stored file

       G = Generated

: F

Show scans (Y or N)?
N

Show arrays (Y or N)?
N

Enter file name: panda


Enter   64 lines of   64 pixels each.

      Enter a  0  or  space for a white pixel.

      Enter a  1  for a black pixel.

      Enter a  *  to quit.

(Missing lines and unfilled line ends default to spaces.)

*   H U F F M A N   C O D I N G   *

Entropy:   1.80

*  OPTIMUM CODES  *

Average code word:   1.86 bits

Code efficiency:     0.97

*  FIXED CODES   *

Average code word:   1.90 bits

Code efficiency:     0.95
```

64 X 64

	HORIZ	VERT	PERIM	QUAD	HILBT	HUFFO	HUFFX
C.R.	1.94	1.77	2.04	1.38	1.79	2.15	2.11
Bits	2112	2320	2008	2976	2288	1908	1943
Runs	261	289	250	367	281		

Figure 4

Chapter 5

ISC System Module Designs

All algorithms and routines in the ISC system are written as functions in the C programming language. The functions were designed and developed with only a moderate regard for processing efficiency. All of the algorithms, except the scanning algorithm for tracing a Hilbert curve [see WIRTH], were originally developed, based on concepts and the processing the algorithm was required to perform.

5.1 Main Program

The main program (function) of the ISC system serves as the user interface into the system and as the driver for causing the user's selected image to be compressed. The main and its primary first level functions are:

main: Prompts the user for the source of the image to be compressed, and whether or not the image scans and arrays are to be displayed. Invokes routines for compressing the image using both run length and Huffman encoding.

scancmprs: Compresses and decompresses the user selected image using each of the five scanning techniques with run length encoding.

Optionally calculates a figure of merit for regionalism for each scanning technique. Also invokes the Huffman coding routines.

scanstats: Displays compression statistics.

5.2 Image Scanning Routines

There are five image scanning routines (functions) in the ISC system; each one follows a different path to scan a bit-mapped binary image represented as an integer array. Each scanning routine generates a sequence of pixel line and position coordinates as it passes through every image pixel, once and only once. The five scanning functions are:

horiz: scans left to right by position, top to bottom by line.

vert: scans top to bottom by line, left to right by position.

perim: scans perimeter, shrinks in.

quad: scans in progressively larger and larger quadrants.

hilbert: scans by tracing a Hilbert curve (a recursive function).

When a scanning function is invoked, the name of the function (encode, decode or merit) that will use the generated line and position coordinates is passed as a parameter. To verify that each scanning routine has indeed been exhaustive, after scanning has been completed, the sum of the run values is checked to ensure that it is equal to the total number of pixels in the original image. While this is not a conclusive test, it does give reasonable assurance.

5.3 Run Length Encoding/Decoding Routines

Run length encoding converts a stream of pixels from an original image into an array of run values. Run length decoding produces the opposite effect, restoring an original image from an array of run values. The ISC system run length encoding and decoding routines are:

encode: Receives a stream of image coordinate (line and position) values from one of the five scanning routines. Accesses the color of the original image pixel at the corresponding location and maintains a running sum as long as successive pixels are the same color. On a change of color, the running sum is placed as a run value into a compressed image array.

The value of the first element of the compressed image array is set to indicate

both the scanning technique being used and the color of the first run. If, at any point, it is determined that the compressed image array will exceed the storage space of the original image, encoding is terminated and the original image is stored unencoded. When the original has been completely scanned, the compression ratio, the number of runs and number of bits in in the compressed image are preserved for later display.

decompress: Based on the value of the first element in a compressed image array, selects the proper scanning technique/routine to decode and restore the original image.

decode: Receives a stream of image coordinate values from one of the five scanning routines while simultaneously accessing an array of run values representing a compressed image. The coordin- are used to set the pixel color at the corresponding location in a bit-mapped array which will hold the reconstructed original image. This array is initially zeroed to effect an all background image and only black runs (every other run value) are used to turn on pixels. Run values and received coordinate values

remain synchronized throughout the decoding process.

5.4 Huffman Coding Routines

Five functions make up the Huffman coding compression routine. Separate sets of optimum and fixed code words are generated, following the coding procedure originally developed by Huffman in 1952 [LYNCH].

huffman: The main Huffman routine totals up the number of occurrences of each 4-pixel block configuration and calls the other routines to construct the optimum and fixed code words and to calculate compression statistics.

hstru: Constructs the Huffman tree-like structure from which the code words can be derived.

showcode: Generates and (optionally) shows the code words; calculates the coding statistics.

gencode: Recursively constructs, displays and calculates the size of the code words.

entropy: Calculates the source entropy with the original image considered as an information source and the 4-pixel blocks as messages.

5.5 Auxiliary Routines

Auxiliary routines provide general purpose functions required by the other processing routines.

merit: Measures the regionalism (as a figure of merit) of a scanning routine by tracking as a deviation, jumps and continued motion in one direction.

scanimage: Creates an original image array from a stream of pixel values (0 or 1) supplied from the terminal keyboard or a stored (image) file.

genimage: Internally generates an original image in response to a user menu selection.

showarray: Displays the contents of an integer array in four column format.

getpixel: Returns a pixel value selected by line and position from an original image array.

putpixel: Sets (to 1) a pixel value selected by line and position in a decompressed image array.

scale: Scales an input or internally generated original image up to the current system preset original image size.

showimage: Displays an original image up to 64 x 64

pixels. Larger images are scaled down to fit.

compare: Compares arrays representing an original and a decompressed image indicating if they are, or are not, identical.

zeroarray: Sets all elements in an image array to zero, in effect setting the image to total background color.

response: Prompts for a user response and checks it for acceptable values.

Chapter 6

Verification and Validation

A variety of binary test images were created and compressed using the ISC system. The test images were first used to verify the correctness of the ISC system algorithms and then to observe the relationships between scanning techniques, scanning densities, run length encoding, Huffman coding, and binary image compression in general. Based on the compression results obtained from a selected group of 10 test images, a number of observations and conjectures have been made and are presented in this chapter. The 10 selected test images and their detailed compression results are provided in Appendix A.

6.1 Test Images

Each of the 10 selected test images was created with a particular base resolution (scanning density) which was the minimum necessary to capture its information content, or 16 x 16 pixels, whichever was greater. Four of the test images were created as external files, the other six are those that can be generated internally by the ISC system.

The 10 selected test images and their base resolutions are:

Name	Description	Base Res.	Source
dog	dog	16 x 16	file
G0	background	16 x 16	generated
G1	one-quarter	16 x 16	generated
G2	one-half	16 x 16	generated
G3	vertical line	16 x 16	generated
G4	quadrant boxes	16 x 16	generated
G5	thick curve	16 x 16	generated
panda	panda	64 x 64	file
text	3 letters	64 x 64	file
flowchart	flowchart	64 x 64	file

In addition to being compressed at their base resolution, each test image was also compressed at higher resolutions, by scaling, up to 128 x 128 pixels. The results provided by the ISC system for four scanning densities (16 x 16, 32 x 32, 64 x 64, and 128 x 128 pixels) have been collected into a single data sheet for each test image and are provided in Appendix A. Data sheets contain the following compression statistics:

(1) Run length encoding (all 5 scanning routines)
    Compression ratio achieved
    Number of bits in the compressed image
    Number of runs in the compressed image
(2) Huffman coding (optimum and fixed codes)
    Entropy of the original image

Compression ratio achieved

Number of bits in the compressed image

Average code word length

Coding efficiency

The compression ratio has been previously defined as the ratio of the bits/pixels in the original image to the bits in the compressed image (see Section 3.3). The entropy of an original image is calculated as the average information content of its 4-pixel blocks when they are considered as messages, and the image itself an information source (see Section 3.3.2). Specifically, if p(i) is the probablility of occurrence of the i-th block configuration, i = 1,..,16, then the entropy H of a binary image is given by:

H = - Summation [p(i) * log2 p(i)] Note: If an image is known to be of solid background color then the probability of any 4-pixel block selected consisting of all background pixels is equal to 1 and the entropy H is then equal to 0 (i.e., no information is conveyed). Conversely, if the image consists of completely random pixels, then the probability of occurrence of each 4-pixel block is equal to 1/16 and the entropy H is equal to 4.00, its maximum.

The average (Huffman) code word length is calculated as follows:

If the frequency of occurrence of each 4-pixel block configuration is p(i), i = 1,...,16, and the code word

selected to represent it is l(i) bits in length then:

average code word length = Summation [p(i) * l(i)]

The coding efficiency is the ratio of the entropy to the average code word length. Since the entropy is the lower bound for the average code word length, the closer the entropy is approached the more efficient the coding scheme.

6.2 Overscan

When an image is scanned with a resolution or scanning density greater than that which is necessary to capture its entire information content (as that information content is perceived by the user), a condition which could be described as overscan occurs. For example, if an original image is that of a triangle, and after scanning at 64 x 64 pixels the scanned representation of the image is correctly perceived by the user to be a triangle, then scanning that same original image at 128 x 128 pixels conveys no additional information but does increase the image's required storage space and transmission time fourfold.

Run length encoding is especially effective in dealing with overscanned images. Not only does run length encoding achieve whatever compression is possible for the image in its base resolution but it can also neutralize up to 100% of the wastefulness caused by overscanning (see data sheets for test images A-8, A-9 and A-10 in Appendix A). When an image is

(over)scanned at twice its base resolution, the number of pixels increases fourfold but the number of runs in the compressed representation of that image, depending upon the image features, may only increase twofold for one-dimensional horizontal and vertical scanning, and may not increase at all for two-dimensional quadrant and hilbert scanning. The principle effect of overscanning is simply larger run values.

6.3 Huffman Coding

The probabilities for generating the fixed Huffman code words used by the ISC system were obtained from the combined frequency distribution of 4-pixel blocks in the three test images that have a 64 x 64 base resolution: panda, text and flowchart. Although these three test images had diverse features, the compression ratios achieved using the fixed code words were within 15% of those achieved with optimum code words.

In contrast to run length encoding, the compression ratio for Huffman coding of image pixel blocks is upper bounded, the bound equal to the size of the block. This occurs because the length of the code word representing each block must be at least 1 bit. Consequently, for the 4-pixel blocks used in this thesis, the maximum compression ratio that can be achieved is 4.00. Further, the upper bound is only achieved when the image to be compressed is a solid color, either all white or all black. In contrast, run length en-

coding can represent a solid color image by a single run value.

Huffman coding also does not handle overscan as well as run length encoding. A fourfold increase in the number of image pixels resulting from a doubling of the scanning density means a fourfold increase in the number of 4-pixel blocks and, if the average length of a code word is already close to 1 bit (a desirable condition), almost a fourfold increase in the size (in bits) of the compressed image.

On the other hand, Huffman coding using optimum code words will do no worse than a compression ratio of 0.80. This results because: the maximum entropy, H, for 4-pixel blocks is 4.00 bits, which occurs when the image pixels are completely random (see Section 3.3.2); and, the average code word length is between H and H+1 bits, a result derived by C. Shannon in his work on information theory. Consequently, the maximum average number of bits to encode each 4-pixel block cannot exceed 5.00, which results in a compression ratio of 0.80. In contrast, for run length encoding, an image with alternating black and white pixels could have a compression ration as low as 0.125 since 8 bits (in the present ISC system implementation) would be required to encode each 1 pixel run.

6.4 Regionalism

An interesting question is how to rate a given scanning technique for use with run length encoding, independent of any particular image. It is intuitively obvious that a scanning technique that traces an unbroken path through an image would tend to stay in a given color region longer than a technique that jumps around and, hence, would have longer runs and achieve a higher compression ratio. Although important, this is not enough. Perimeter scanning traces an unbroken curve (and so would horizontal scanning if pixels were scanned on the return trip) but it does not perform as well for overscanned images as quadrant scanning which does not. A second intuitive quality factor of a good scanning technique is regionalism: the ability to exhaust a region as the scan spreads across the image. Hilbert scanning is the only one of the five techniques that has both features: tracing an unbroken curve and regionalism and hilbert scanning had the best overall performance of the five scanning techniques (see Figures 5a and 5b).

While it is relatively straightforward to check for an unbroken curve, how does one measure regionalism? A simple algorithm, called "merit," which takes both features in account, has been developed for the ISC system. The algorithm is based on the following two premises:

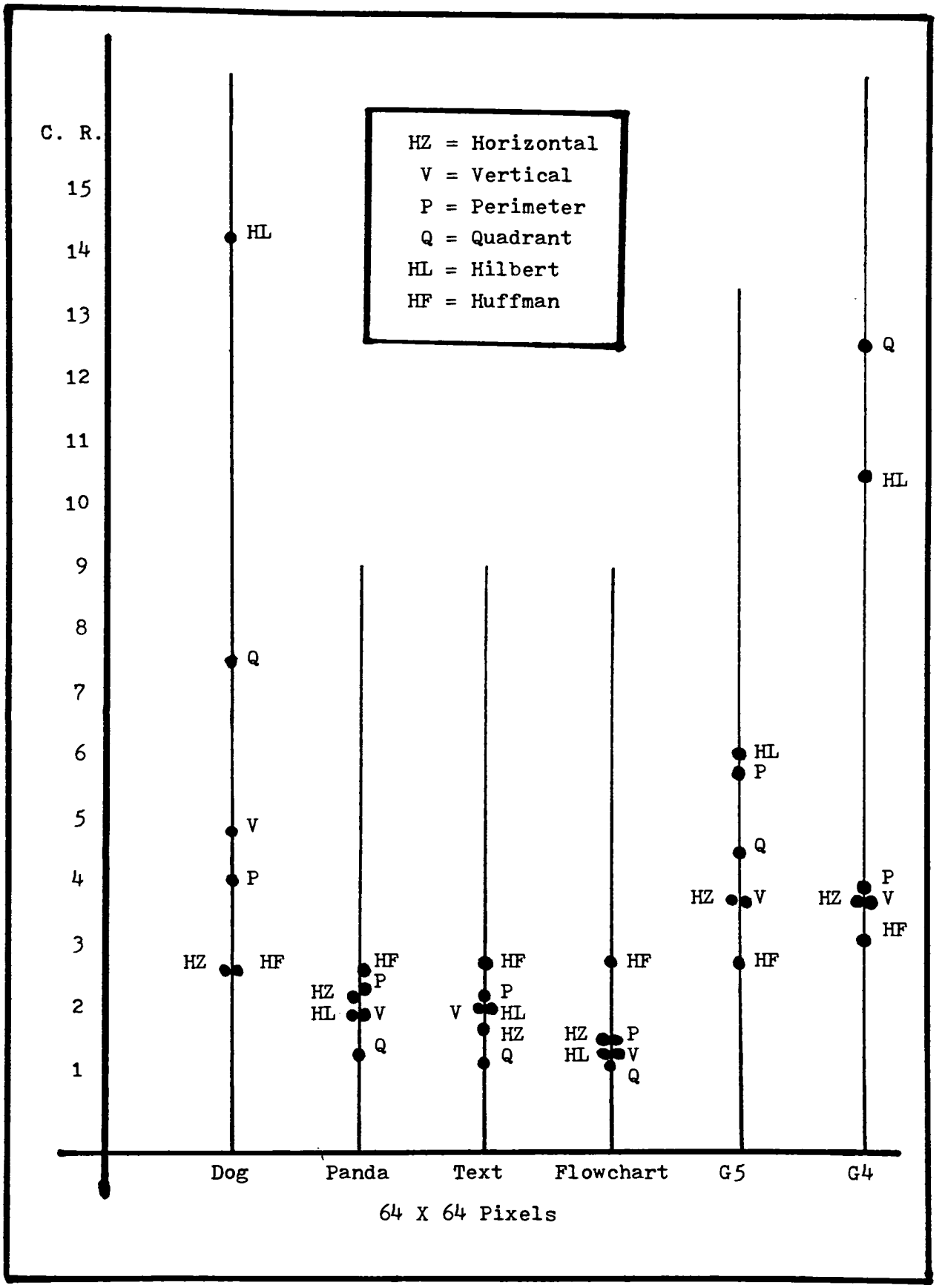

C. R.
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
HZ = Horizontal
V = Vertical
P = Perimeter
Q = Quadrant
HL = Hilbert
HF = Huffman
HL
Q
V
P
HZ
HF
HF
P
HZ
HL
V
Q
HF
P
V
HL
HZ
Q
HF
HZ
P
HL
V
Q
HL
P
Q
HZ
V
HF
Q
HL
P
HZ
V
HF
Dog
Panda
Text
Flowchart
G5
G4
64 X 64 Pixels

Figure 5a

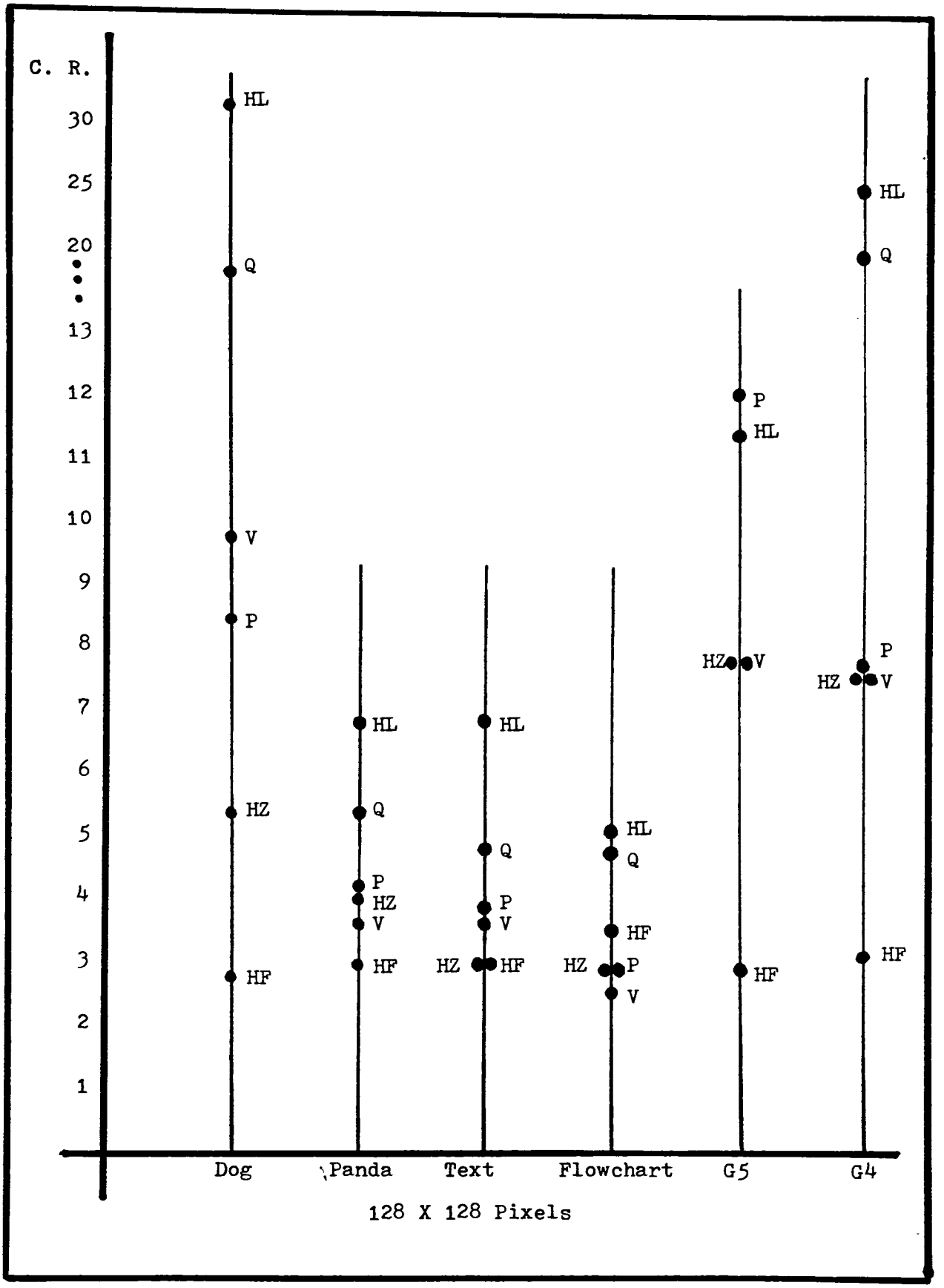

C. R.
30
25
20
13
12
11
10
9
8
7
6
5
4
3
2
1
HL
Q
V
P
HZ
HF
HL
Q
P
HZ
V
HF
HL
Q
P
V
HZ
HF
HL
Q
HF
HZ
P
V
P
HL
HZ
V
HF
HL
Q
P
HZ
V
HF
Dog
Panda
Text
Flowchart
G5
G4
128 X 128 Pixels

Figure 5b

(1) A scanning technique that keeps scanning in the same direction, even on an unbroken path, will not remain long in the same region.

(2) When a scanning technique jumps, the further the jump, the more likely the landing point will be outside the current region.

The merit algorithm measures regionalism by monitoring the path of a scanning technique and imposing penalties for continued motion in one direction and for jumps, the penalty proportional to the size of the jump. It does this by maintaining a non-decreasing deviation count which is incremented by one for continued motion in one direction, and incremented by the size of any jump (less one if the jump is opposite the previous direction). X and Y (position and line) coordinates are evaluated independently but contribute to the same deviation count. A figure of merit is calculated by dividing the final deviation count by the total number of pixel traversals which is the total number of pixels minus one. The lower the figure of merit the greater the regionalism.

The merit algorithm was applied to each of the five scanning techniques and the results were in general agreement with the results obtained from the compression of the test images, that is, the techniques with the lower figures of merit were the more effective in regard to compression (see Figure 5b). Specifically, the figures of merit obtained for

each scanning technique at a 128 x 128 pixel scanning density were:

Scanning Technique	Figure of Merit
Horizontal	1.97
Vertical	1.97
Perimeter	0.98
Quadrant	0.98
Hilbert	0.50

Figures of merit at a 64 x 64 scanning density were only slightly different from those at 128 x 128 pixels. Somewhat surprising is the relationships between the figures of merit: the perimeter and quadrant techniques had identical figures of merit and these were almost exactly half that of horizontal and vertical scanning and almost exactly twice that of hilbert scanning. It is surprising since each method, except horizontal and vertical which were identical, accumulated its deviation count (over 16,000 for a figure of merit of 0.98 at 128 x 128) in totally different ways (e.g., perimeter scanning never having a jump and quadrant scanning having numerous jumps of various sizes).

Chapter 7

Conclusions

The conclusions expressed in this thesis are primarily based on observations made in scanning and compressing the test images previously described. While they hold for the test images they may not hold for binary images in general.

7.1 Overall Conclusions

Run length encoding is probably the simplest technique to implement for the compression of binary graphics images. It is especially effective for minimizing the storage requirements of binary graphics images that have relatively large regions of solid color (either black or white), such as charts, posters, drawings, templates, logos, etc.

Performance for run length encoding deteriorates rapidly as an image acquires more and more detail. Consequently, it would not work well with lines of text except at relatively high scanning densities. As image detail increases, the number of image regions increases also, but the average region size (in pixels) decreases. A point can be quickly reached where it is more efficient to simply leave the image uncompressed. An image compression system that utilizes run length encoding should be designed to break off encoding (as

is done in the ISC system) at the point where it is no longer possible for the compression ratio to exceed one.

There is no question that when different scanning techniques are used with run length encoding, different degrees of compression will result. A better question is "how different?" Based on the results of compressing the more detailed test images, those with a 64 x 64 pixel base resolution (panda, text and flowchart), it appears that, when the scanning density is equal to the image's base resolution, there is not a significant difference among the five scanning techniques; the compression ratios achieved are somewhat clustered and all five are outperformed by Huffman coding of 4-pixel blocks (see Chapter 6, Figure 5a). However, as the scanning density increases beyond the image's base resolution, simulating an overscanned image, the compression ratios distinctly separate (see Chapter 6, Figure 5b). The one-dimensional scanning techniques, horizontal and vertical, are clearly outperformed by the two-dimensional scanning techniques, quadrant and hilbert. Hilbert scanning turns in the best performance of all, and all five scanning techniques outperform Huffman coding.

Ideally, the scanning density for an image should be just high enough to capture all of its details or information content as perceived by the user. This presents a problem when there is "mixed bag" of images. So as not to suffer any

loss of information, the scanning density would have to be set to accommodate the most detailed image in the group and many images may be overscanned. Run length encoding, by absorbing the redundant pixels of overscanned images as larger run values, can cut the wasteful effects of overscan by up to 50% with one-dimensional scanning techniques, and up to 100% with two-dimensional scanning techniques, depending upon image features.

As can be seen from the data sheets for the most detailed of the test images (A-1, A-8, A-9 and A-10 in Appendix A), the number of runs for hilbert scanning remains constant as the scanning density is increased. The same also holds true for quadrant scanning but hilbert scanning achieves the higher compression ratios. The constant number of runs is primarily due to the fact that the scanning density is always being increased by powers of 2 and hilbert scanning scans in a quadrant-like manner (similar to quadrant scanning but without the jumps).

The above observation of constancy of runs would not quite hold true for actual scanners and real images. Increasing the scanning density at the edges of regions of a real image results in different pixel configurations (due to the finer resolution) than scaling up a test image that has already been resolved into pixels. However, this would be the case only at a region's edges, not within a region, so

the less image detail, and the more rectangular an image's features, the more closely real life scanning will reflect the results of scanning the test images.

A very desirable attribute of an image compression scheme would be that the total number of bits required to encode an image would be independent of the scanning density used. Hilbert scanning linked to run length encoding comes closer to achieving this goal than any of the other four scanning techniques. Hilbert scanning also has the best figure of merit for regionalism (as measured by the ISC system's merit algorithm) and, consequently, it seems reasonable that new scanning techniques could be evaluated as to their effectiveness with run length encoding by computing their figure of merit and not having to scan a myriad of images.

While two-dimensional hilbert and quadrant scanning techniques are more effective than one-dimensional techniques for compressing overscanned images, their ISC system implementations are processing intensive. This could limit their usefulness in responding to the time constraints of real-time applications; however, it may not be so great a problem. Since the transmission of images via facsimile is usually limited to 9600 baud, the extra processing time may not be noticed. Also, compression of images for storage can usually be done off-line, as a background task, so again the extra processing time may not matter.

Perimeter scanning, while something of a hybrid between one and two dimensional scanning techniques, can be a very effective for compressing certain classes of binary images. Because of edge distortion on video screens, and because of the potential for skewing of paper documents as they are scanned, most screen and paper images have wide borders or margins of background color. Perimeter scanning exploits this image feature the best of the five scanning techniques. As an example, if an 8 1/2 x 11 inch document with one inch borders is perimeter scanned at 100 pixels per inch the first run value obtained from run length encoding would be greater than 300,000.

When fixed code words were used to Huffman code the 64 x 64 base resolution images, the compression ratios achieved were within 15% of those achieved with optimum code words. Consequently, it appears that if fixed code words are derived from a well chosen set of typical documents, the relatively small reduction in the compression ratio achieved would be well worth eliminating the need to generate and store optimum code words with each compressed image.

Neither run length encoding or Huffman coding would be of any practical use for compressing a binary image of completely random pixels. A random image itself would already

be the minimum configuration for the information it contains.

7.2 Additional Remarks

Throughout this thesis, images and scanner have been assumed to be "perfect," that is, image regions were always assumed to have sharp, distinct edges and unrestricted increases in scanning density were allowed by scaling. In real life situations, as the scanning density is increased, a point will eventually be reached where region edges become jagged and feathered, and "noise" (false detail) begins to appear in background regions. At this point run length encoding would quickly become unusable (as more and more small runs are encountered), unless appropriate "smoothing" measures are taken.

Another area recognized but not given a thorough treatment in this thesis is data transmission errors. While errors are rare when data is transferred to and from computer storage they are a common problem due to channel noise when data is transmitted. For uncompressed images a few inverted pixels is usually not a serious problem. However, what would be an acceptable error bit rate for uncompressed image data, would probably not be acceptable for that same data in compressed form. Run length encoding is particularly error sensitive. A single high order bit inversion in a large run value could greatly distort or even totally destroy the resulting decompressed image. A lost run value would not

only distort an image but would also reverse its polarity from that point onward. One way to minimize the effects of transmission errors is to confine them to as small an area as possible by periodically transmitting synchronization points in the compressed image, for example, at the end of each line for horizontal scanning. Use of synchronization points, however, reduces the degree of compression that could otherwise be achieved.

7.3 Future Enhancements and Directions

The ISC system in its present configuration is primarily a research tool for the investigation of binary image compression. With little modification it could also serve as the foundation for an actual "production" image compression system. The following system enhancements would be helpful in achieving this end.

A useful enhancement of the ISC system would be to extend it to rectangular images since most real world images are not square. The only system components that really depend on square images are the quadrant and Hilbert curve scanning techniques. The extension could probably be most easily accomplished by embedding the rectangular image in a minimum square and ignoring pixels outside the extents. Even if outside pixels are not ignored, if they are always treated as background color pixels, the compression achieved by run length encoding would probably not be effected much since the

entire perimeter of almost all images is composed of background pixels and the entended portion of the square would then simply be a continuation of the current run.

Another possible enhancement for the ISC system would be to follow the CCITT standard and to generate Huffman code words for the run values obtained from run length encoding. This could significantly increase the degree of compression that can be achieved for detailed images (which have smaller average run values) since the number of bits required to store a run value would then be closer to its actual magnitude.

Although tracing a Hilbert curve appears to be a very attractive scanning technique, its recursive implementation in the ISC system is complex and processsing intensive. An excellent ISC system enhancement would be to develop an equivalent implementation based on an incremental algorithm, assuming that one is possible.

Finally, the ideal ISC system would automatically select the best of the five scanning techniques (the one with the highest compression ratio) to compress any given image. In its present configuration the ISC system can only determine the best scanning technique after having tried all five techniques to compress the image. An image or pattern recognition capability for the ISC system, perhaps based on artifi-

cial intelligence, would greatly enhance its potential development into a system for practical application.

Glossary

base resolution: the minimum scanning density necessary to capture an image's total information content.

binary image: a digital image whose pixels can only assume two values, white, usually represented by 0, and black, represented by 1; also known as a bi-level or two-tone image.

bit-mapped image: every pixel in the image is mapped to one bit in memory.

entropy: a measure of the average information content of a message from an information source.

pel: same as a pixel.

pixel: the smallest resolvable point of a picture (a picture element).

overscan: a condition where an image has been scanned with a scanning density greater than that which is necessary to capture its entire information content.

quadtree: a tree-like data structure generated by recursively decomposing an image into quadrants such that each leaf (terminal node) of the tree represents a region of a single color, white or black.

quantization: converting a continuous analog signal to a fixed set of discrete magnitude levels.

run: a sequence of pixels of the same color.

run value: the total number of pixels in a run.

Bibliography

Bharath, R., "Information Theory," BYTE, December 1987.

CCITT, "Telegraph and Telematic Services Equipment: Recommendation of the S and T Series," Volume VII, Fascicle VII.2 (Yellow Book), VIIth Plenary Assembly, Geneva, November 1980.

Corbin, H., "An Introduction to Data Compression," BYTE, April 1981.

Davies, E. and Plummer, A., "A New Method for the Compression of Binary Picture Data," Proceedings of the IEEE 5th International Conference on Pattern Recognition, 1980.

Ekstrom, M. P., "Digital Image Processing Techniques," Academic Press, 1984.

Held, G., "Data Compression," John Wiley & Sons, 1983.

Kunt, M. and Johnsen, O., "Block Coding of Graphics: A Tutorial Review," Proceedings of the IEEE, Vol. 68, No. 7, July 1980.

Lauzon, J., Mark, D., Kikuchi, L. and Guevara, J., "Two-Dimensional Run-Encoding for Quadtree Representation," Computer Vision, Graphics, and Image Processing, Vol. 30, 1985.

Lynch, T., "Data Compression, Techniques and Applications," Van Nostrans Reinhold, 1985.

Pountain, D., "Run-Length Encoding," BYTE, June 1987.

Pracht, B. and Bowyer, K., "Adaptations of Run-Length Encoding for Image Data," IEEE, 1987.

Samet, H. and Rosenfeld, A., "Quadtree Representation of Binary Images, IEEE 5th International Conference on Pattern Recognition, 1980.

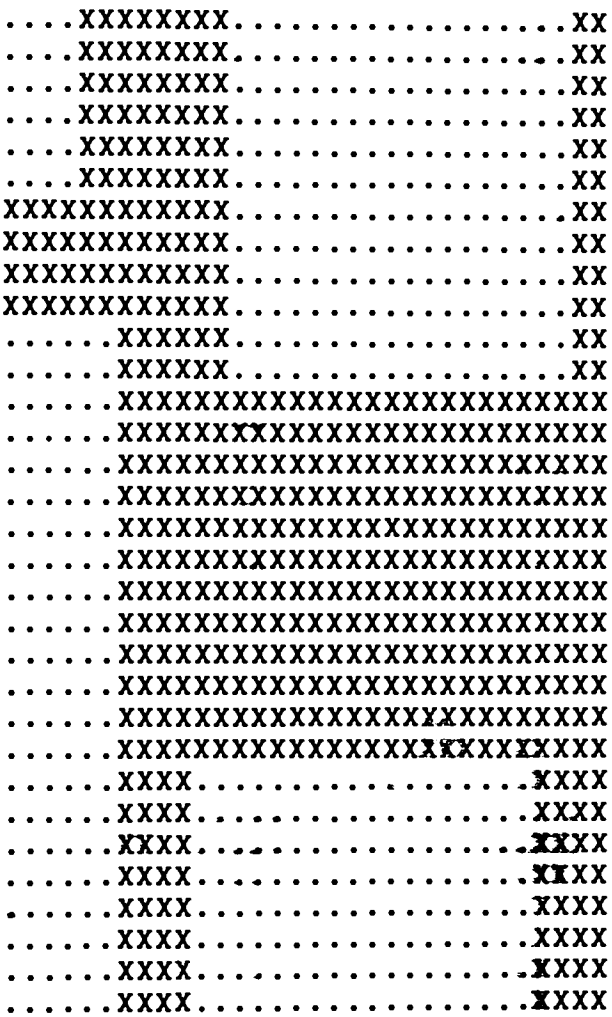
Ting, D. and Prasada, B., "Digital Processing Techniques for Encoding of Graphics," Proceedings of the IEEE, Vol. 68, No. 7, July 1980.

Walter, G., "Redundancy Reduction and Data Compaction Technology in Microform Image Transmission Systems," Journal of Micrographics, October 1982.

Wirth, N., "Algorithms + Data Structures = Programs," Prentice-Hall, 1976.

A P P E N D I X A

Test Images and Data Sheets

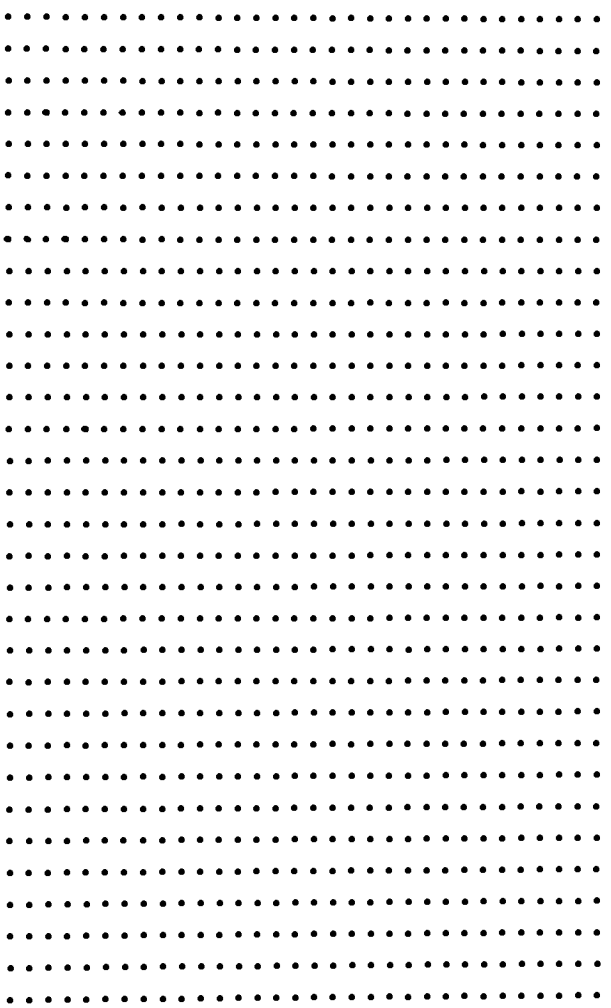


Dog

I S C S Y S T E M D A T A S H E E T

I M A G E : d o g

		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.	--	1.32	2.65	5.32
	Bits	--	776	1544	3080
	Runs	--	96	192	384
VERTICAL	C. R.	1.19	2.42	4.70	9.62
	Bits	216	424	872	1704
	Runs	26	52	104	208
PERIMETER	C. R.	--	2.03	4.10	8.36
	Bits	--	504	1000	1960
	Runs	--	62	122	242
QUADRANT	C. R.	--	2.10	7.42	17.50
	Bits	--	488	552	936
	Runs	--	60	60	60
HILBERT	C. R.	1.14	4.57	14.22	31.03
	Bits	224	224	288	528
	Runs	27	27	27	27
HUFFMAN					
Entropy		2.37	1.60	1.00	1.00
Optimum Codes					
	C. R.	1.61	2.26	2.67	2.67
	Bits	159	454	1532	6128
	Avg Code	2.48	1.77	1.50	1.50
	Efficiency	95%	90%	67%	67%
Fixed Codes					
	C. R.	1.16	1.93	2.67	2.67
	Bits	220	530	1532	6128
	Avg Code	3.44	2.07	1.50	1.50
	Efficiency	69%	77%	67%	67%

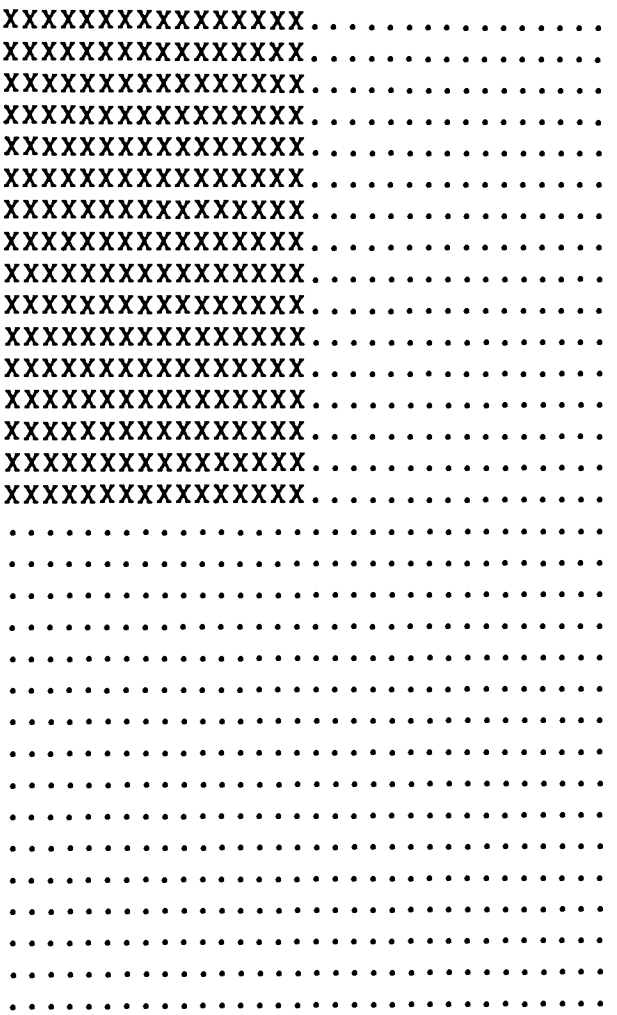


GO

I S C S Y S T E M D A T A S H E E T

I M A G E : G0 (Background)

		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.	8.00	32.00	128.00	512.00
	Bits	32	32	32	32
	Runs	1	1	1	1
VERTICAL	C. R.	8.00	32.00	128.00	512.00
	Bits	32	32	32	32
	Runs	1	1	1	1
PERIMETER	C. R.	8.00	32.00	128.00	512.00
	Bits	32	32	32	32
	Runs	1	1	1	1
QUADRANT	C. R.	8.00	32.00	128.00	512.00
	Bits	32	32	32	32
	Runs	1	1	1	1
HILBERT	C. R.	8.00	32.00	128.00	512.00
	Bits	32	32	32	32
	Runs	1	1	1	1
HUFFMAN					
Entropy		0.00	0.00	0.00	0.00
Optimum Codes					
	C. R.	4.00	4.00	4.00	4.00
	Bits	64	256	1024	4096
	Avg Code	1.00	1.00	1.00	1.00
	Efficiency	0%	0%	0%	0%
Fixed Codes					
	C. R.	4.00	4.00	4.00	4.00
	Bits	64	256	1024	4096
	Avg Code	1.00	1.00	1.00	1.00
	Efficiency	0%	0%	0%	0%



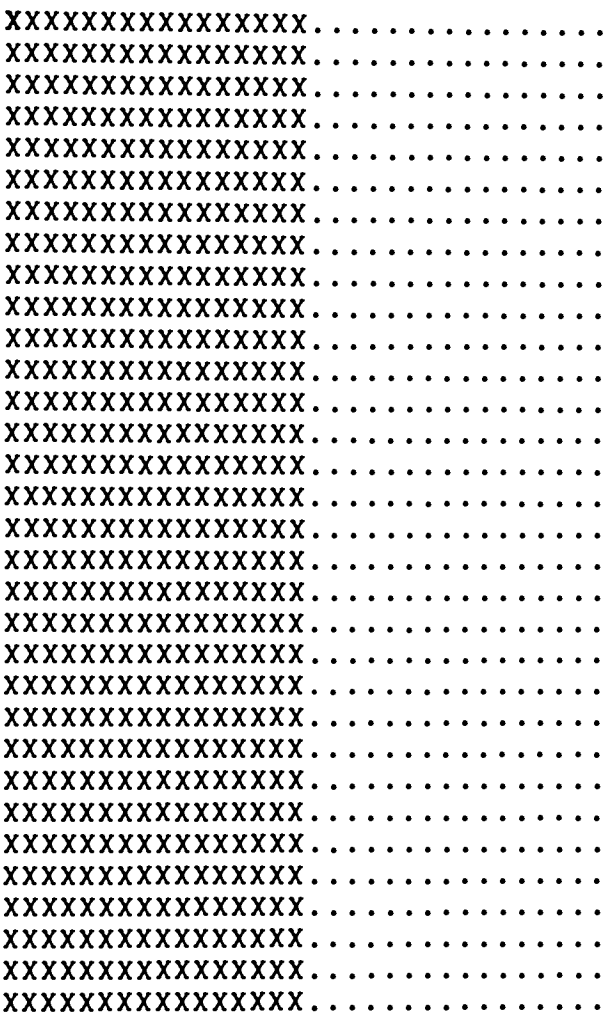
G1

A-3

I S C S Y S T E M D A T A S H E E T

I M A G E : G1 (Quarter square)

		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.	1.88	3.66	7.64	15.63
	Bits	136	280	536	1048
	Runs	16	32	64	128
VERTICAL	C. R.	1.88	3.66	7.64	15.63
	Bits	136	280	536	1048
	Runs	16	32	64	128
PERIMETER	C. R.	1.88	3.88	7.88	11.98
	Bits	136	264	520	1368
	Runs	16	32	64	128
QUADRANT	C. R.	10.67	18.29	73.14	292.57
	Bits	24	56	56	56
	Runs	2	2	2	2
HILBERT	C. R.	8.00	12.80	51.20	204.80
	Bits	32	80	80	80
	Runs	3	3	3	3
HUFFMAN					
Entropy		0.81	0.81	0.81	0.81
Optimum Codes					
	C. R.	3.20	3.20	3.20	3.20
	Bits	80	320	1280	5120
	Avg Code	1.25	1.25	1.25	1.25
	Efficiency	65%	65%	65%	65%
Fixed Codes					
	C. R.	3.20	3.20	3.20	3.20
	Bits	80	320	1280	5120
	Avg Code	1.25	1.25	1.25	1.25
	Efficiency	65%	65%	65%	65%

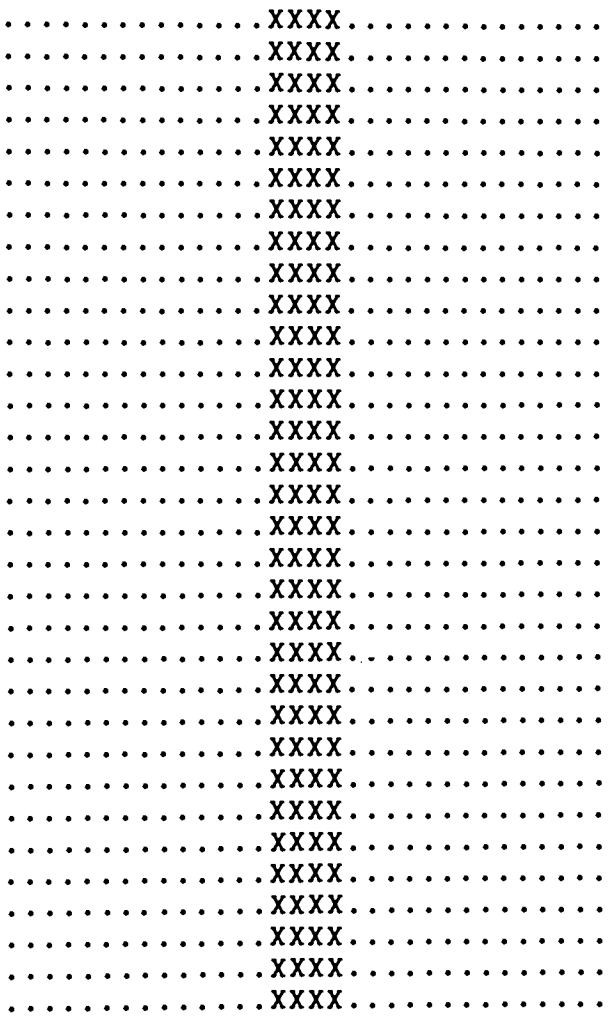


G2

I S C S Y S T E M D A T A S H E E T

I M A G E : G2 (Half square)

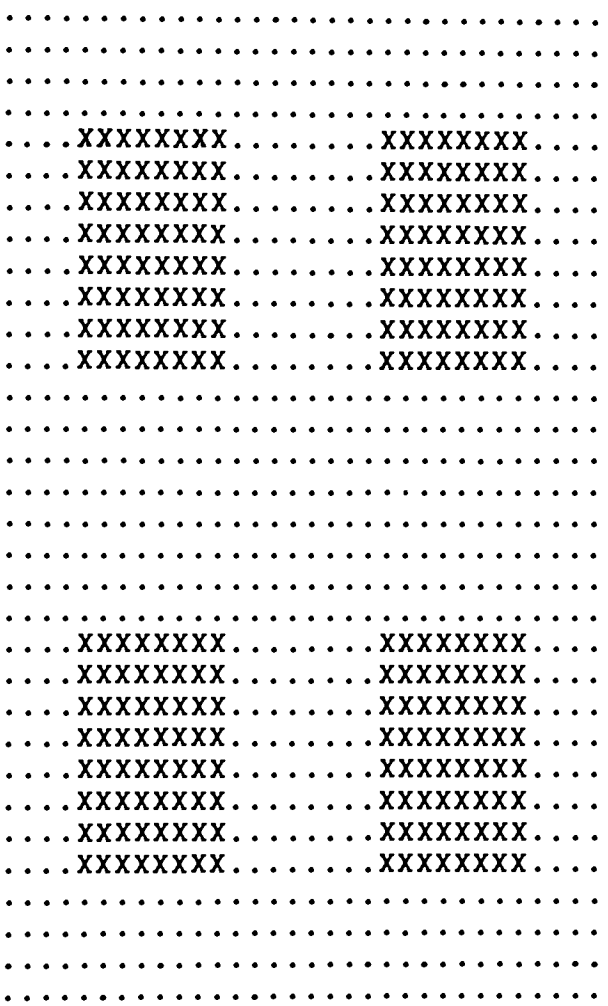
		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.	--	1.97	3.97	7.97
	Bits	--	520	1032	2056
	Runs	--	64	128	256
VERTICAL	C. R.	10.67	18.29	73.14	292.57
	Bits	24	56	56	56
	Runs	2	2	2	2
PERIMETER	C. R.	1.78	3.76	7.76	15.75
	Bits	144	272	528	1040
	Runs	17	33	65	129
QUADRANT	C. R.	6.40	9.85	39.38	157.54
	Bits	40	104	104	104
	Runs	4	4	4	4
HILBERT	C. R.	8.00	12.80	51.20	204.80
	Bits	32	80	80	80
	Runs	3	3	3	3
HUFFMAN					
Entropy		1.00	1.00	1.00	1.00
Optimum Codes					
	C. R.	2.67	2.67	2.67	2.67
	Bits	96	384	1536	6144
	Avg Code	1.50	1.50	1.50	1.50
	Efficiency	67%	67%	67%	67%
Fixed Codes					
	C. R.	2.67	2.67	2.67	2.67
	Bits	96	384	1536	6144
	Avg Code	1.50	1.50	1.50	1.50
	Efficiency	67%	67%	67%	67%


.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....
.....XXXX.....

I S C S Y S T E M D A T A S H E E T

I M A G E : G3 (Vertical line)

		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.	--	1.94	3.94	7.94
	Bits	--	528	1040	2064
	Runs	--	65	129	257
VERTICAL	C. R.	8.00	16.00	51.20	204.80
	Bits	32	64	80	80
	Runs	3	3	3	3
PERIMETER	C. R.	1.03	2.17	4.45	8.94
	Bits	248	472	920	1832
	Runs	30	58	114	226
QUADRANT	C. R.	--	2.06	6.74	22.26
	Bits	--	496	608	736
	Runs	--	61	61	61
HILBERT	C. R.	1.33	5.33	16.00	56.89
	Bits	192	192	256	288
	Runs	23	23	23	23
HUFFMAN					
Entropy		1.50	1.06	0.54	0.54
Optimum Codes					
	C. R.	2.29	2.91	3.56	3.56
	Bits	112	352	1152	4608
	Avg Code	1.75	1.38	1.13	1.13
	Efficiency	86%	77%	48%	48%
Fixed Codes					
	C. R.	1.33	2.00	3.56	3.56
	Bits	192	512	1152	4608
	Avg Code	3.00	2.00	1.13	1.13
	Efficiency	50%	53%	48%	48%

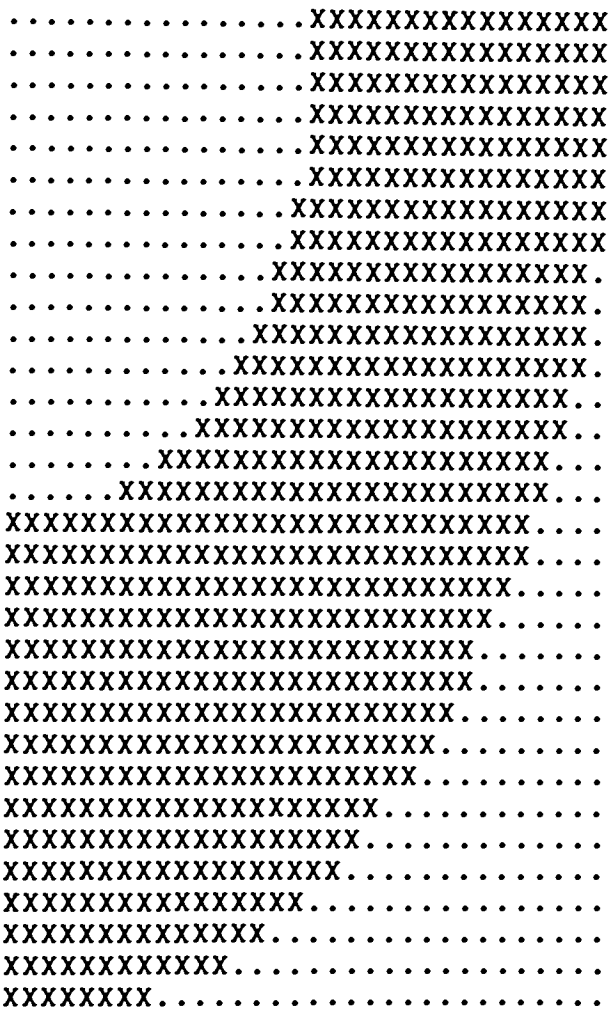


G4

I S C S Y S T E M D A T A S H E E T

I M A G E : G4 (Squares in quadrants)

		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.	--	1.88	3.76	7.76
	Bits	--	544	1088	2112
	Runs	--	65	129	257
VERTICAL	C. R.	--	1.88	3.76	7.76
	Bits	--	544	1088	2112
	Runs	--	65	129	257
PERIMETER	C. R.	--	1.88	3.82	7.82
	Bits	--	544	1072	2096
	Runs	--	65	129	257
QUADRANT	C. R.	--	3.76	12.80	20.48
	Bits	--	272	320	800
	Runs	--	33	33	33
HILBERT	C. R.	1.23	4.92	10.67	26.95
	Bits	208	208	384	608
	Runs	25	25	25	25
HUFFMAN					
Entropy		1.50	0.81	0.81	0.81
Optimum Codes					
	C. R.	2.29	3.20	3.20	3.20
	Bits	112	320	1280	5120
	Avg Code	1.75	1.25	1.25	1.25
	Efficiency	86%	65%	65%	65%
Fixed Codes					
	C. R.	1.33	3.20	3.20	3.20
	Bits	192	320	1280	5120
	Avg Code	3.00	1.25	1.25	1.25
	Efficiency	50%	65%	65%	65%



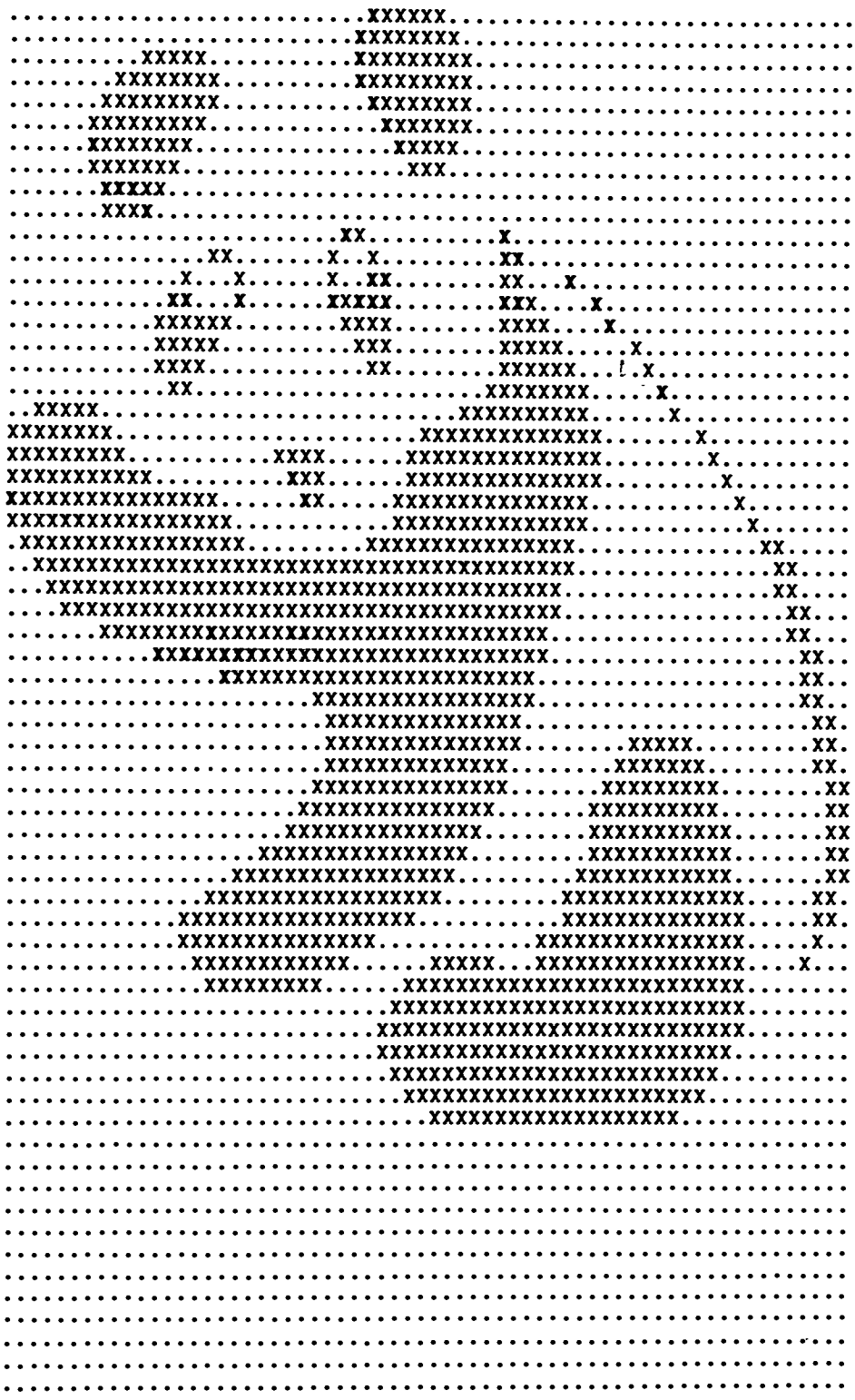
G5

A-7

I S C S Y S T E M D A T A S H E E T

I M A G E : G5 (Thick curve)

		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.	--	1.94	3.94	7.94
	Bits	--	528	1040	2064
	Runs	--	65	129	257
VERTICAL	C. R.	--	1.94	3.94	7.94
	Bits	--	528	1040	2064
	Runs	--	65	129	257
PERIMETER	C. R.	1.39	2.98	5.89	12.12
	Bits	184	344	696	1352
	Runs	22	42	84	166
QUADRANT	C. R.	1.78	2.91	4.49	8.53
	Bits	144	352	912	1920
	Runs	17	43	103	215
HILBERT	C. R.	2.46	4.13	6.17	11.57
	Bits	104	248	664	1416
	Runs	12	30	72	156
HUFFMAN					
Entropy		1.81	1.56	1.36	1.21
Optimum Codes					
	C. R.	2.15	2.36	2.55	2.67
	Bits	119	433	1609	6134
	Avg Code	1.86	1.69	1.57	1.50
	Efficiency	97%	92%	86%	81%
Fixed Codes					
	C. R.	1.80	2.06	2.24	2.36
	Bits	142	498	1830	6939
	Avg Code	2.22	1.95	1.79	1.69
	Efficiency	81%	80%	76%	71%

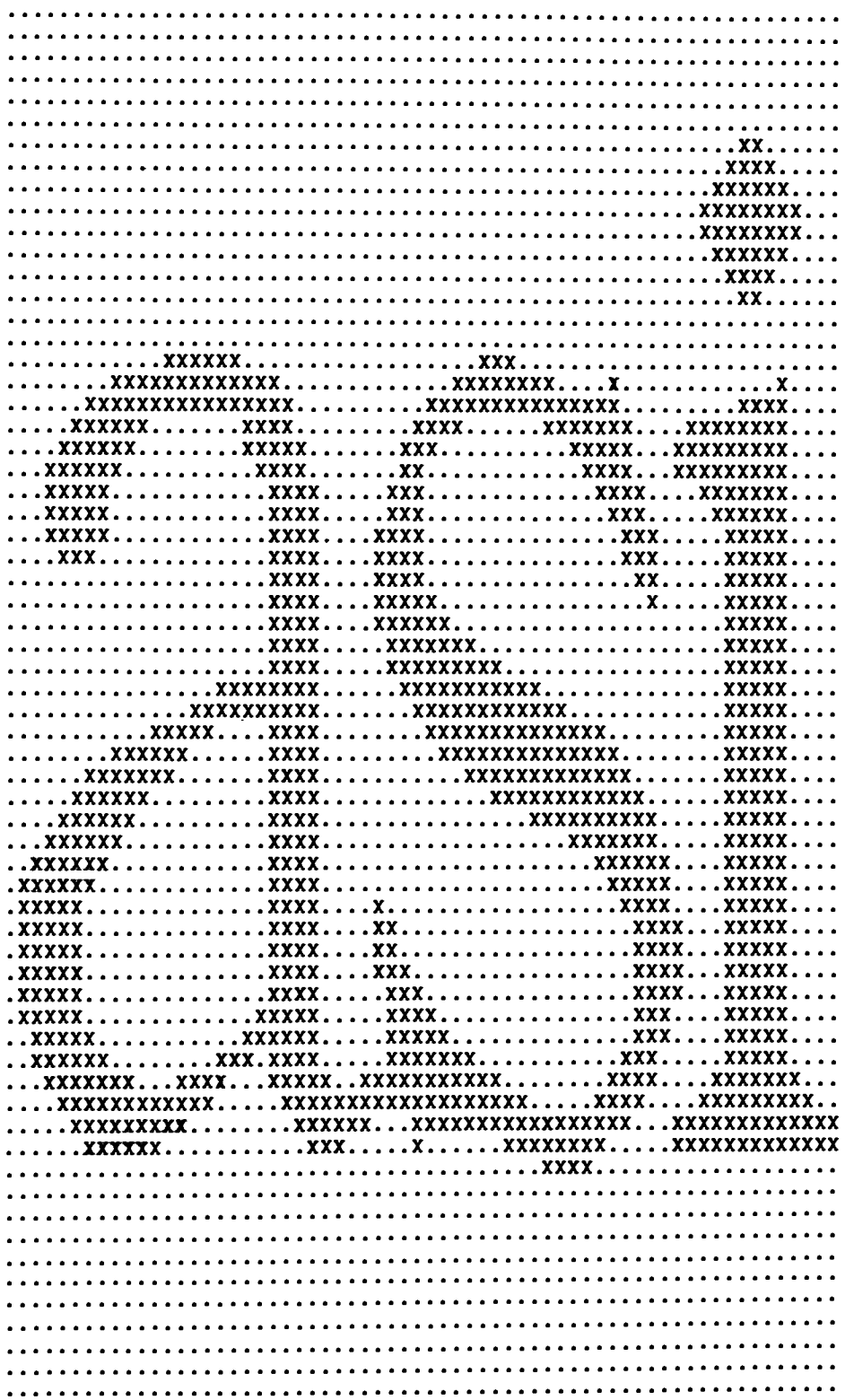


Panda

I S C S Y S T E M D A T A S H E E T

I M A G E : p a n d a

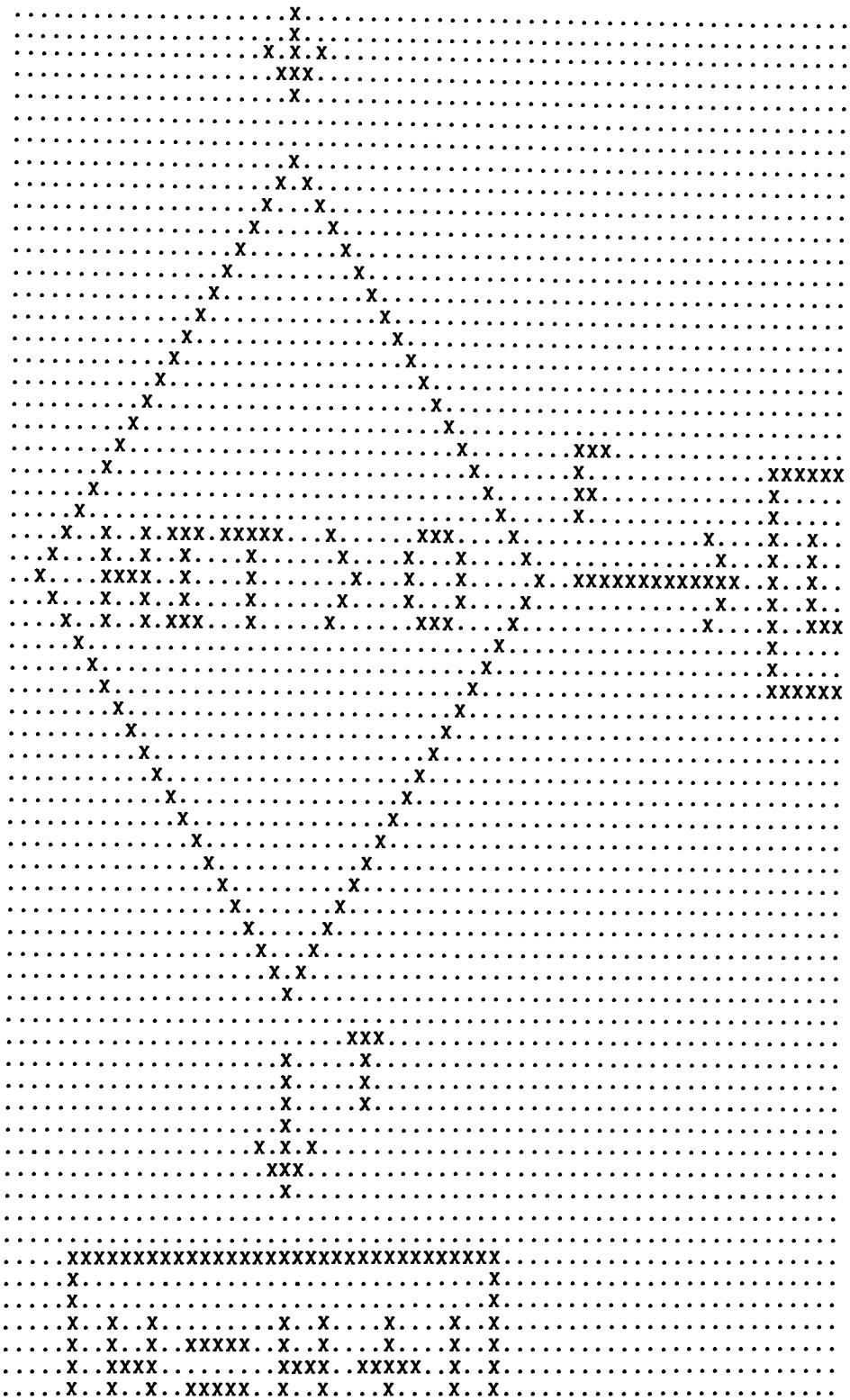
		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.			1.94	3.91
	Bits			2112	4192
	Runs			261	521
VERTICAL	C. R.			1.77	3.54
	Bits			2320	4624
	Runs			289	577
PERIMETER	C. R.			2.04	4.06
	Bits			2008	4040
	Runs			250	502
QUADRANT	C. R.			1.38	5.22
	Bits			2976	31.36
	Runs			367	367
HILBERT	C. R.			1.79	6.65
	Bits			2288	2464
	Runs			281	281
HUFFMAN					
Entropy				1.80	1.18
Optimum Codes					
	C. R.			2.15	2.85
	Bits			1908	5754
	Avg Code			1.86	1.40
	Efficiency			97%	84%
Fixed Codes					
	C. R.			2.11	2.67
	Bits			1943	6142
	Avg Code			1.90	1.50
	Efficiency			95%	79%



I S C S Y S T E M D A T A S H E E T

I M A G E : t e x t

		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.			1.50	3.01
	Bits			2736	5440
	Runs			337	673
VERTICAL	C. R.			1.74	3.48
	Bits			2352	4704
	Runs			293	585
PERIMETER	C. R.			1.86	3.68
	Bits			2200	4456
	Runs			274	546
QUADRANT	C. R.			1.19	4.63
	Bits			3440	3536
	Runs			425	425
HILBERT	C. R.			1.75	6.65
	Bits			2336	2464
	Runs			289	289
HUFFMAN					
Entropy				1.69	1.09
Optimum Codes					
	C. R.			2.22	2.98
	Bits			1845	5500
	Avg Code			1.80	1.34
	Efficiency			94%	81%
Fixed Codes					
	C. R.			2.14	2.76
	Bits			1910	5926
	Avg Code			1.87	1.45
	Efficiency			91%	75%



Flowchart

I S C S Y S T E M D A T A S H E E T

I M A G E : f l o w c h a r t

		16 x 16	32 x 32	64 x 64	128 x 128
HORIZONTAL	C. R.			1.34	2.67
	Bits			3056	6144
	Runs			381	761
VERTICAL	C. R.			1.24	2.47
	Bits			3312	6640
	Runs			413	823
PERIMETER	C. R.			1.32	2.62
	Bits			3104	6256
	Runs			387	773
QUADRANT	C. R.			1.17	4.55
	Bits			3488	3600
	Runs			429	429
HILBERT	C. R.			1.23	4.83
	Bits			3328	3392
	Runs			405	405
HUFFMAN					
Entropy				1.28	0.70
Optimum Codes					
	C. R.			2.54	3.31
	Bits			1612	4944
	Avg Code			1.57	1.21
	Efficiency			82%	58%
Fixed Codes					
	C. R.			2.18	2.87
	Bits			1882	5714
	Avg Code			1.84	1.40
	Efficiency			70%	50%

A P P E N D I X B

ISC System Source Code

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

Definitions for ISC System constants

*****************************************************************************/


#define VPIX 64                     /*  vertical image pixels     */
#define HPIX 64                     /*  horizontal image pixels   */
#define ISIZE 8                     /*  word/integer size         */
#define SCANSIZE 64                 /*  scan image side size      */
#define SHOWSIZE 64                 /*  show image max side size  */
#define MAXVALUE 255                /*  maximum word value        */

#define VHPIX (VPIX*HPIX)           /*  total image pixels        */
#define XPIX  HPIX                  /*  pixels per square side    */

#define V VPIX                      /*  vertical image words      */
#define H (HPIX/ISIZE)              /*  horizontal image words    */
#define VH (V*H)                    /*  total image words         */

#define NO  0
#define YES 1

#define ABS(X)     ((X) < 0 ? -(X) : (X))  /*  Absolute value macro  */
#define LOGB(N,B)  (log(N)/log(B*1.0))     /*  LogB(N) macro         */
```

```
#include <stdio.h>
#include "scan.h"

/*************************************************************************

            IMAGE SCANNING AND COMPRESSION (ISC) SYSTEM

                              May 1, 1988

*************************************************************************/


                                /*   External variables              */
float CR[7];                    /*       Compression ratio           */
int   BT[7];                    /*       Bits per compressed image   */
int   RN[5];                    /*       Runs per compressed image   */

int   showscans;                /*       Show pixels as scanned      */
int   showarrays;               /*       Show image arrays           */


/*************************************************************************

Function:  main

Purpose:  To provide a user interface to the ISC system and to
          compress a selected original image using run length
          Huffman encoding.

*************************************************************************/

main()
{
   int I[V*H];                      /* Original image array         */
   int imageno,i;
   char select,option;
   char filein[20];
   char response();

   extern int showscans;
   extern int showarrays;

   FILE *in;                       /*  File pointer for terminal   */
                                   /*     or stored file image     */
   showscans = NO;
   showarrays = NO;
                           /*  Determine source of original image */

printf("\n\n                IMAGE SCANNING AND COMPRESSION\n\n\n");
printf("Select the means of providing the target image:\n\n");
printf("       S = Scanner or keyboard\n\n");
printf("       F = Stored file\n\n");
printf("       G = Generated\n\n");
   fflush(stdout);
   select = response("SFG");
   printf(": %c\n",select);
```

```
    fflush(stdout);

                                /*  Display pixels as scanned? */

    printf("\nShow scans (Y or N)? \n");
    fflush(stdout);
    option = response("NY");
    printf("%c\n",option);
    fflush(stdout);
    if (option == 'Y')
        showscans = YES;

                              /*  Display original and compressed */
                              /*    image arrays?                 */

    printf("\nShow arrays (Y or N)? \n");
    fflush(stdout);
    option = response("NY");
    printf("%c\n",option);
    fflush(stdout);
    if (option == 'Y')
        showarrays = YES;

    if (select == 'S')
        {
         in = stdin;              /* Original image will originate */
         scanimage(I,in);         /*    from a terminal or scanner */
        }
    else if (select == 'F')
          {
           printf("\nEnter file name: "); /* Original image will */
           fflush(stdout);       /* originate from a stored file  */
           scanf("%s",filein);
           printf("%s\n",filein);
           if ((in = fopen(filein,"r")) != NULL)
                scanimage(I,in);
           else
               {
                printf("No such file: %s\n",filein);
                exit();
               }
          }
    else                          /* Original image will be selected */
          {                       /*   and generated internally      */

           printf("\nSelect image number 0 thru 5: ");
           printf("\n\n    0 : Total white.");
           printf("\n\n    1 : One quadrant black.");
           printf("\n\n    2 : Half black, half white.");
           printf("\n\n    3 : Black vertical line.");
           printf("\n\n    4 : Black squares in quadrants.");
           printf("\n\n    5 : Black quarter circle.\n");
           fflush(stdout);
           scanf("%d",&imageno);
           printf("%d\n",imageno);
           genimage(imageno,I);
```

```
        }

   showimage(I);     /*  Show original image up to 64 X 64 display */

   scancmprs(I);     /*  Scan and compress original image          */

   scanstats();      /*  Show compression statistics               */

}

/*************************************************************************

Function: scancmprs

Purpose:  To scan and compress the original image using each of the
          five scanning techniques and run length encoding.  Also,
          to compress the original image using an application of
          Huffman coding.  And, when activated, to determine the
          degree of regionalism exhibited by any scanning technique.

*************************************************************************/

scancmprs(I)
   int I[V*H];                          /* Original image array     */
   {
    int C[VH+6];                        /* Compressed image array   */
    int D[V*H];                         /* Decompressed image array */
    int encode();
    int merit();

    /*  For all five scanning techniques:    */
    /*          Compress   (I ---> C)        */
    /*          Decompress (C ---> D)        */
    /*          Compare    (D <--> I)        */

    horiz(encode,I,C);
    decompress(C,D);
    compare(D,I);

    vert(encode,I,C);
    decompress(C,D);
    compare(D,I);

    perim(encode,I,C);
    decompress(C,D);
    compare(D,I);

    quad(encode,I,C);
    decompress(C,D);
    compare(D,I);

    hilbert(encode,I,C);
    decompress(C,D);
    compare(D,I);
```

```
    huffman(I);

/**************************

    horiz(merit,I,C);           When activated, determines the
    vert(merit,I,C);            degree of regionalism in any
    perim(merit,I,C);           scanning technique, expressed
    quad(merit,I,C);            as a figure of merit.
    hilbert(merit,I,C);

**************************/

   }

/*****************************************************************************

Function: scanstats

Purpose:  To display, in tabular form, the compression statistics
          from both run length and Huffman encoding.

*****************************************************************************/

scanstats()
  {
   int i;

printf("\n\n\n                                          %d  X  %d",VPIX,HPIX);
printf("\n\n           ");
printf("HORIZ     VERT     PERIM     QUAD     HILBT     HUFFO     HUFFX");
printf("\n         -----");
printf("-------------------------------------------------------------------");
printf("\nC.R. ");
   for (i = 0; i < 7; i++)
        printf("    %6.2f",CR[i]);
   printf("\n\nBits ");
   for (i = 0; i < 7; i++)
        printf("    %6d",BT[i]);
   printf("\n\nRuns ");
   for (i = 0; i < 5; i++)
        printf("    %6d",RN[i]);
   printf("\n");
  }
```

```
#include <stdio.h>
#include "scan.h"

/*************************************************************************

Function: horiz

Purpose: To scan every pixel of an original image (represented by
         an integer array) from left to right by position, and from
         top to bottom by line.

*************************************************************************/

horiz(nd,A,B)
   int (*nd)();                  /* Encode, decode or merit function */
   int A[];                      /* Original image array */
   int B[];                      /* Compressed image array */
   {
    extern int showscans;

    int code = 1;               /* Processing control code  */
    int line, pos;
    int encode();
    int merit();

    if ((showscans && *nd == encode) || *nd == merit)
        printf("\n\n*   H O R I Z O N T A L   S C A N   *\n");

    for (line = 0; line < VPIX; line++)       /* Generate and pass   */
         {                                    /*   line and position */
          if (showscans && *nd == encode)     /*   covering entire   */
              printf("\n");                   /*   image space       */
          for (pos = 0; pos < HPIX; pos++)
               (*nd)(A,B,line,pos,&code);
         }
    code = -1;                        /* Processing complete code value */
    (*nd)(A,B,line,pos,&code);
   }

/*************************************************************************

Function: vert

Purpose: To scan every pixel of an original image (represented by
         an integer array) from top to bottom by line, and from
         left to right by position.

*************************************************************************/

vert(nd,A,B)
   int (*nd)();                  /* Encode, decode or merit function */
   int A[];                      /* Original image array */
   int B[];                      /* Compressed image array */
   {
    extern int showscans;
    int line, pos;
```

```
    int code = 2;              /* Processing control code */
    int encode();
    int merit();

    if ((showscans && *nd == encode) || *nd == merit)
        printf("\n\n*   V E R T I C A L   S C A N   *\n");

    for (pos = 0; pos < HPIX; pos++)          /* Generate and pass   */
         {                                    /*   line and position */
          if (showscans && *nd == encode)     /*   covering entire   */
              printf("\n");                   /*   image space       */
          for (line = 0; line < VPIX; line++)
               (*nd)(A,B,line,pos,&code);
         }
    code = -1;                      /* Processing complete code value */
    (*nd)(A,B,line,pos,&code);
   }

/*************************************************************************

Function: perim

Purpose: To scan every pixel of an original image (represented by
         an integer array) by scanning the image perimeter and
         shrinking in.

*************************************************************************/

perim(nd,A,B)
   int (*nd)();                  /* Encode, decode or merit function */
   int A[];                      /* Original image array */
   int B[];                      /* Compressed image array */
   {
    extern int showscans;

    int line, pos;
    int code = 3;              /* Processing control code */
    int pmin = 0;
    int pmax = HPIX-1;
    int lmin = 0;
    int lmax = VPIX-1;
    int encode();
    int merit();

    if ((showscans && *nd == encode) || *nd == merit)
        printf("\n\n*   P E R I M E T E R   S C A N   *\n\n");

    while ((lmin <= lmax) && (pmin <= pmax))   /* Continue scanning  */
                               /*  and chopping off an edge of the */
                               /*  perimeter until nothing is left */
          {
           for (line = lmin,pos = pmin;lmin <= lmax,pos <= pmax;pos++)
                (*nd)(A,B,line,pos,&code);
           ++lmin;
           if (showscans && *nd == encode)
               printf("\n");
```

```
            for (line = lmin,pos = pmax;pmin <= pmax,line <= lmax;line++)
                 (*nd)(A,B,line,pos,&code);
            --pmax;
            if (showscans && *nd == encode)
                printf("\n");

            for (line = lmax,pos = pmax;lmin <= lmax,pos >= pmin;pos--)
                 (*nd)(A,B,line,pos,&code);
            --lmax;
            if (showscans && *nd == encode)
                printf("\n");

            for (line = lmax,pos = pmin;pmin <= pmax,line >= lmin;line--)
                 (*nd)(A,B,line,pos,&code);
            ++pmin;
            if (showscans && *nd == encode)
                printf("\n");
           }
    code = -1;                          /* Processing complete code value */
    (*nd)(A,B,line,pos,&code);
   }

/*************************************************************************

Function: quad

Purpose: To scan every pixel of an original image (represented by an
         integer array) in progressively larger and larger quadrants.
         Image is assumed to be square with sides a power of 2.

*************************************************************************/

quad(nd,A,B)
   int (*nd)();                     /* Encode, decode or merit function */
   int A[];                         /* Original image array */
   int B[];                         /* Compressed image array */
   {
    extern int showscans;
    unsigned i,set,mask;
    int line,pos;
    int code = 4;                   /* Processing control code */
    int encode();
    int merit();

    if ((showscans && *nd == encode) || *nd == merit)
        printf("\n\n*   Q U A D R A N T   S C A N   *\n");

    set = VPIX*VPIX/2;              /*  Highest bit of VPIX*VPIX - 1  */

    for (i = 0; i < VPIX*VPIX; i++)         /*  Consecutive numbers  */
                                            /*    of all pixels      */
         {
          if (showscans && *nd == encode && (i % HPIX == 0))
              printf("\n");
          line = pos = 0;           /* Select out odd and even bits  */
```

```
          mask = set;                /*   of pixel sequence number as */
          while (mask > 1)           /*   separate numbers            */
               {
                line = (i & mask) ? 2*line+1 : 2*line;   /* Odd bits */
                mask /= 2;
                pos = (i & mask) ? 2*pos+1 : 2*pos;      /* Even bits */
                mask /= 2;
               }
          (*nd)(A,B,line,pos,&code);
         }
    code = -1;                       /* Processing complete code value */
    (*nd)(A,B,line,pos,&code);
   }

/*************************************************************************

Function: hilbert

Purpose: To scan every pixel of an original image (represented by an
         integer array) in by following the path of a Hilbert curve.
         Image is assumed to be square with sides a power of 2.

*************************************************************************/

hilbert(nd,A,B)
   int (*nd)();                  /* Encode, decode or merit function */
   int A[];                      /* Original image array */
   int B[];                      /* Compressed image array */
   {
    extern int showscans;
    int line = VPIX-1;
    int pos = HPIX-1;
    int order = 0;
    int code = 5;               /* Processing control code */
    int x = VPIX;
    int encode();
    int merit();

    if ((showscans && *nd == encode) || *nd == merit)
        printf("\n\n*   H I L B E R T   S C A N   *\n\n");

    while (x > 1)                /* Establish order of Hilbert curve */
          {
           x /= 2;
           ++order;
          }

    (*nd)(A,B,line,pos,&code); /* Start off recursive function calls */
    pl(order,nd,A,B,&line,&pos,&code);

    code = -1;                       /* Processing complete code value */
    (*nd)(A,B,line,pos,&code);
   }

pl(order,nd,A,B,line,pos,code)               /* lst pattern function */
   int order,(*nd)(),A[],B[],*line,*pos,*code;
```

```
    {
     int encode();

     if (order > 0)
          {
           p4(order-1,nd,A,B,line,pos,code);
              --(*pos);
             (*nd)(A,B,*line,*pos,code);
           p1(order-1,nd,A,B,line,pos,code);
              --(*line);
             (*nd)(A,B,*line,*pos,code);
           p1(order-1,nd,A,B,line,pos,code);
              ++(*pos);
             (*nd)(A,B,*line,*pos,code);
           p2(order-1,nd,A,B,line,pos,code);
           if (showscans && *nd == encode)
               printf("\n");
          }
    }

p2(order,nd,A,B,line,pos,code)                  /* 2nd pattern function */
    int order,(*nd)(),A[],B[],*line,*pos,*code;
    {
     if (order > 0)
          {
           p3(order-1,nd,A,B,line,pos,code);
              ++(*line);
             (*nd)(A,B,*line,*pos,code);
           p2(order-1,nd,A,B,line,pos,code);
              ++(*pos);
             (*nd)(A,B,*line,*pos,code);
           p2(order-1,nd,A,B,line,pos,code);
              --(*line);
             (*nd)(A,B,*line,*pos,code);
           p1(order-1,nd,A,B,line,pos,code);
          }
    }

p3(order,nd,A,B,line,pos,code)                  /* 3rd pattern function */
    int order,(*nd)(),A[],B[],*line,*pos,*code;
    {
     if (order > 0)
          {
           p2(order-1,nd,A,B,line,pos,code);
              ++(*pos);
             (*nd)(A,B,*line,*pos,code);
           p3(order-1,nd,A,B,line,pos,code);
              ++(*line);
             (*nd)(A,B,*line,*pos,code);
           p3(order-1,nd,A,B,line,pos,code);
              --(*pos);
             (*nd)(A,B,*line,*pos,code);
           p4(order-1,nd,A,B,line,pos,code);
          }
    }
```

```
p4(order,nd,A,B,line,pos,code)                /* 4th pattern function */
   int order,(*nd)(),A[],B[],*line,*pos,*code;
   {
    if (order > 0)
        {
         p1(order-1,nd,A,B,line,pos,code);
            --(*line);
           (*nd)(A,B,*line,*pos,code);
         p4(order-1,nd,A,B,line,pos,code);
            --(*pos);
           (*nd)(A,B,*line,*pos,code);
         p4(order-1,nd,A,B,line,pos,code);
            ++(*line);
           (*nd)(A,B,*line,*pos,code);
         p3(order-1,nd,A,B,line,pos,code);
        }
   }

/*************************************************************************

Function: zigzag

Purpose: To scan every pixel of an original image (represented by an
         integer array) in a zig-zag manner.  The image is assumed to
         be square with sides a power of 2.

*************************************************************************/

zigzag(nd,A,B)
   int (*nd)();                  /* Encode, decode or merit function */
   int A[];                      /* Original image array */
   int B[];                      /* Compressed image array */
   {
    extern int showscans;

    int code = 1;               /* Processing control code */
    int i,j,k;
    int encode();
    int merit();

    if ((showscans && *nd == encode) || *nd == merit)
        printf("\n\n*   Z I G - Z A G   S C A N   *\n");

    for (i = 0; i < VPIX; i++)                /* Generate and pass   */
         {                                    /*   line and position */
          if (showscans && *nd == encode)     /*   covering entire   */
              printf("\n");                   /*   image space       */
          for (j = 0; j <= i; j++)
               if (i % 2)
                    (*nd)(A,B,j,i-j,&code);
               else (*nd)(A,B,i-j,j,&code);
         }
    for (i = 1; i < VPIX; i++)
         {
          if (showscans && *nd == encode)
              printf("\n");
```

```
          k = i;
          for (j = VPIX-1; j >= i; j--)
               if (i % 2)
                    (*nd)(A,B,j,k++,&code);
               else (*nd)(A,B,k++,j,&code);
        }
     code = -1;                    /* Processing complete code value */
     (*nd)(A,B,j,k,&code);
    }
```

```
#include <stdio.h>
#include <math.h>
#include "scan.h"

/*****************************************************************************

Function: huffman

Purpose: To generate Huffman code words for an image array with
         blocks of 4 contiguous horizontal pixels considered as
         messages of an information source.  Optimum codes words
         are generated from the actual block frequency distri-
         bution and sub-optimum code words are generated using
         a fixed approximate frequency distribution.  Compression
         ratios, source entropy, average bits per coded message
         and compression efficiency are also calculated.

*****************************************************************************/

huffman(I)
   int I[VH];                          /* Original image array  */
   {
    extern int   showscans;            /* True ==>  show code words   */
    extern int   showarrays;           /* True ==>  show frequency array  */

    extern float CR[];                 /* Compression Ratio           */
    extern int   BT[];                 /* Bits in compressed image    */

    float evalue;
    float entropy();

    int i,A[32],B[32],C[32];           /* Arrays for Huffman algorithm */

    static int D[32] = { 70,  4,  2,  2,   /* Frequency distribution */
                          2,  0,  0,  2,   /*    for generation of   */
                          2,  0,  0,  0,   /*    fixed codes.        */
                          2,  0,  2, 12 }; /*    (sum = 100)         */

    if (showscans || showarrays)
        printf("\f\n*   H U F F M A N   C O D I N G   *\n");

    zeroarray(A,32);
    for (i = 0; i < VH; i++)           /* Determine frequency distribution */
         {                             /*    of blocks in image            */
          ++A[I[i] >> 4 & 017];
          ++A[I[i] & 017];             /* 017 octal = 00001111 binary */
         }
    if (showarrays)
        showarray(A,16,"A");

    evalue = entropy(A,16,2*VH);       /* Calculate source entropy */
    printf("\n\nEntropy: %6.2f\n",evalue);

    if (showscans)
        printf("\n\n\n*  OPTIMUM CODES  *\n");
```

```
    hstruc(A,B,C,2*VH);                /* Generate optimum code words */

    showcode(A,B,C,2*VH,evalue,6);     /* Show optimum code words     */

    if (showscans)
        printf("\n\n\n*  FIXED CODES  *\n");

    hstruc(D,B,C,100);         /* Generate sub-optimum code words from */
                               /*    fixed frequency distribution      */

    showcode(A,B,C,2*VH,evalue,7);     /* Show fixed code words       */
   }

/*************************************************************************

Function: hstruc

Purpose: To create a tree-like array structure from which the
         Huffman code words for a given frequency distribution
         of 4 pixel image blocks can be generated.

*************************************************************************/

hstruc(A,B,C,total)
   int A[32];                    /* Frequency nodes                */
   int B[32],C[32];              /* Parent/child links             */
   int total;                    /* Total number of block samples */
   {
    int i;
    int min = 0;
    int node = 15;
    int index1 = 0;
    int index2 = 0;

    zeroarray(B,32);
    zeroarray(C,32);

    while (min <= total)      /* Apply Huffman procedure until */
          {                   /*   root node generated*/

           min = total+1;           /* Search for node with minimum */
           for (i = 0; i <= node; i++)  /*   unprocessed frequency */
                {
                 if ((A[i] < min) && (B[i] == 0))
                     {
                      min = A[i];
                      index1 = i;
                     }
                }
           B[index1] = 1;
           min = total+1;           /* Search for next smallest node */
           for (i = 0; i <= node; i++)
                {
                 if ((A[i] < min) && (B[i] == 0))
                     {
                      min = A[i];
```

```
                        index2 = i;
                      }
                  }
            A[++node] = A[index1] + A[index2];/* Combine frequencies */
                                              /*   & create new node */
            B[index1] = B[index2] = node;     /* Save relationships  */
            C[index1] = 0;                    /*   between parent &  */
            C[index2] = 1;                    /*   children nodes    */
           }
   }

/*****************************************************************************

Function: showcode

Purpose: To display the Huffman code words generated for the
         given frequency distribution of the image pixel blocks
         and calculate the compression ratio for the image.

*****************************************************************************/

showcode(A,B,C,total,evalue,ss)
   int A[],B[],C[],total,ss;
   float evalue;
   {
    extern int showscans;
    int i,k;
    int sum = 0;

    for (i = 0; i < 16; i++)
         {
          if (showscans)
              printf("\n\n%4d:  %5d\t\t",i,A[i]);
          k = 0;
          if (A[i] < total)
              gencode(A,B,C,i,total,&k);
          else
              {                              /* Single color image */
               k = 1;
               if (showscans)
                   printf("%ld",1);
              }
          if (showscans)
              printf("\t\t\t    %4d",A[i]*k);
          sum = sum + A[i]*k;
         }
    printf("\n\nAverage code word: %6.2f bits\n",(sum*1.0)/total);
    printf("\n\nCode efficiency:   %6.2f %\n",evalue*total/(sum*1.0));
    CR[ss-1] = (VHPIX)*1.0/sum;
    BT[ss-1] = sum;
   }

/*****************************************************************************

Function: gencode
```

```
Purpose: To generate the Huffman code words by recursively (to
         print in reverse order) going from each leaf node to
         the root.

*****************************************************************************/

gencode(A,B,C,i,total,k)
   int A[],B[],C[],i,total,*k;
   {
    extern int showscans;
    if (A[i] < total)
        {                              /* Traverse tree from leaf to root */
         gencode(A,B,C,B[i],total,k);
         if (showscans)
             printf("%ld",C[i]);
         ++(*k);
        }
   }

/*****************************************************************************

Function: entropy

Purpose: To calulate the entropy of the Huffman coding process
         with each of the 4-pixel block configurations considered
         to be a message from an information source.

*****************************************************************************/

float entropy(A,size,total)
   int A[];
   int size,total;
   {
    float h = 0.0;
    float p;
    int i;

    for (i = 0; i < size; ++i)
         {
          if (A[i] > 0)
              {
               p = (A[i]*1.0)/total;
               h += -(p * LOGB(p,2));
              }
         }
    return(h);
   }
```

```
#include <stdio.h>
#include <math.h>
#include "scan.h"


/*****************************************************************************

Function: encode

Purpose: To run length encode a stream of pixel values from an
         image (array) by line and position as determined (passed)
         by one of a set of scanning routines.  The resulting
         run values (integer) array represents the compressed
         image.

*****************************************************************************/

encode(I,C,ln,ps,cd)
   int I[VH];                       /* Original image array      */
   int C[VH+6];                     /* Compressed image array    */
   int ln, ps;                      /* Image pixel line/position */
   int *cd;                         /* Processing code           */
   {
    extern int   showscans;         /* True ==>  show pixel stream     */
    extern int   showarrays;        /* True ==>  show compressed array */

                                    /* Variables to collect statistics */
    extern float CR[];              /*    Compression Ratio            */
    extern int   BT[];              /*    Bits in compressed image     */
    extern int   RN[];              /*    Number of runs/run values    */

    int i;                          /* Local processing variables */
    unsigned bit;
    static unsigned current;
    static int indx;
    static int run;
    static int sum;
    static int runcount;
    static int ss;

    switch (*cd)
    {
     case 1 :                                 /* Horizontal scanning */
     case 2 :                                 /* Vertical scanning   */
     case 3 :                                 /* Perimeter scanning  */
     case 4 :                                 /* Quadrant scanning   */
     case 5 :                                 /* Hilbert scanning    */

              run = sum = runcount = indx = 0;
              bit = getpixel(I,HPIX,ln,ps); /* Get first image pixel */
              current = bit;
              if (current)                  /* Set first array element */
                   C[indx] = *cd * 2;       /*    with scan technique  */
              else C[indx] = *cd * 2 - 1;   /*    and first color      */
              ss = *cd - 1;
              *cd = 0;
```

```
            /* fall through to case 0 */

     case 0 :                                  /* Main processing */

            bit = getpixel(I,HPIX,ln,ps);      /* Get image pixel by   */
                                               /*    line and position */
            if (showscans)
                printf("%ld",bit);

            if (bit == current)
                ++run;                         /* Continue current run */
            else
                {                              /* End current run */
                 if (run > MAXVALUE)
                     {
                      C[++indx] = 0;             /* Use multiple array */
                      C[++indx] = run / MAXVALUE; /*   elements   */
                      C[++indx] = run % MAXVALUE;
                     }
                 else C[++indx] = run;
                 sum += run;
                 run = 1;                        /* Start new run */
                 current = bit;
                 ++runcount;
                }
            if (indx > VH)                     /* If too many runs, */
                *cd = -2;                      /*    stop encoding  */
            break;

    case -1 : /* Last pixel scanned, finish up */

            if (run > MAXVALUE)                /* End last run */
                {
                 C[++indx] = 0;
                 C[++indx] = run / MAXVALUE;
                 C[++indx] = run % MAXVALUE;
                }
            else C[++indx] = run;
            sum += run;
            C[++indx] = -1;                /* Add array terminator */
            ++runcount;
            if (indx > VH)
                {
                 printf("\n\nImage not encoded \n");
                 C[0] = 0;              /* Encoded image would exceed */
                 for (indx = 0; indx < VH; )   /*   original image */
                      {                        /*   size, so image */
                       i = indx;               /*   is not encoded */
                       C[++indx] = I[i];
                      }
                 C[++indx] = -1;
                 sum = VHPIX;
                 runcount = 0;
                }
            if (showarrays)
                {                          /* Show compressed array */
```

```
                printf("\n\n");
                showarray(C,indx+1,"C");
               }
            if (sum == VHPIX)              /* Verify sum of runs    */
                                           /*   equals total pixels */
               {
            CR[ss] = (VHPIX)*1.0/(indx*ISIZE)*1.0;/* Compression Ratio */
                BT[ss] = indx*ISIZE;            /* Total bits output */
                RN[ss] = runcount;              /* Runs/run values   */
               }
            else
               {
                printf("\n\n=====>  Incorrect sum %d  <=====\n\n",sum);
                CR[ss] = 0.0;
                BT[ss] = 0;
                RN[ss] = 0;
               }
            break;
    case -2 :             /* No encoding */
            break;
    }
   }

/*************************************************************************

Function: decompress

Purpose: To select the proper scanning technique to decode a
         compressed image, restoring the image as a bit-mapped
         (integer) array.

*************************************************************************/

decompress(C,D)
   int C[VH+6];                    /* Compressed image array */
   int D[VH];                      /* Array for decompressed image */
   {
    int i;
    int decode();

    switch ((C[0]+1)/2)            /* Scanning technique derived  */
    {                              /*    from first array element */
     case 0 :
              for (i = 0; i < VH; ++i)
                   D[i] = C[i+1];
              break;

     case 1 :
              horiz(decode,C,D);
              break;

     case 2 :
              vert(decode,C,D);
              break;

     case 3 :
```

```
              perim(decode,C,D);
              break;

     case 4 :
              quad(decode,C,D);
              break;

     case 5 :
              hilbert(decode,C,D);
              break;
    }
  }

/*************************************************************************

Function: decode

Purpose: To decode/decompress an array of run values representing a
         compressed image, restoring the image as a bit-mapped
         (integer) array.

*************************************************************************/

decode(C,D,ln,ps,cd)
   int C[VH+2];                  /* Compressed image array */
   int D[VH];                    /* Decompressed image array */
   int ln, ps;
   int *cd;                      /* Processing condition code */
   {
    int i;                       /* Local processing variables */
    static int current;
    static int indx;
    static int run;

    switch (*cd)
    {
     case 1 :                          /* Horizontal scanning */
     case 2 :                          /* Vertical scanning   */
     case 3 :                          /* Perimeter scanning  */
     case 4 :                          /* Quadrant scanning   */
     case 5 :                          /* Hilbert scanning    */

              zeroarray(D,VH);         /* Set image array to background */
              indx = 0;
              run = C[indx];
              if (run % 2)             /* Determine color of first */
                   current = 0;        /*   run value              */
              else current = 1;
              run = C[++indx];         /* Get first run value */
              if (run == 0)
                  {
                   run = C[++indx] * MAXVALUE;   /* Multiple element */
                   run += C[++indx];             /*    run value     */
                  }
              *cd = 0;
              /* falls through to case 0 */
```

```
     case 0 :                                  /* Main processing */
             if (run == 0)
                 {                       /* Run finished, get next run value */
                  run = C[++indx];
                  if (run == 0)
                      {
                       run = C[++indx] * MAXVALUE;   /* Multi-byte run */
                       run += C[++indx];             /*   value        */
                      }
                  current = 1 - current;       /* Reverse current color */
                 }
             if (current)
                 putpixel(D,ln,ps);      /* Put black image pixels only */
             --run;
             if (run < 0)              /* Runs ended before image complete */
                 {
                  printf("\n\n  Compressed array ended first\n\n");
                  *cd = -2;
                 }
             break;

   case -1 : /* Last image pixel restored */
             break;
   case -2 : /* Processing following decompression, if any */
             break;
    }
   }

/*************************************************************************

Function: merit

Purpose: To measure the degree of regionalism of a scanning technique
         and calculate a corresponding figure of merit.

*************************************************************************/

merit(A,B,y,x,cd)
   int A[],B[];                    /* Dummy array variables */
   int y,x;                        /* y (line) and x (position) of scan */
   int *cd;                        /* Processing condition code */
   {
    static int xh,xdir,yh,ydir;    /* Last coordinate and direction  */
    static int dev;                /* Non-decreasing deviation count */


    if (*cd >= 0)
        {                          /* Code ==> track deviation */
         if (*cd > 0)              /* Code ==> first position  */
             {
              dev = 0;             /* Initialize variables */
              xh = x;
              xdir = 0;
              yh = y;
              ydir = 0;
```

```
             *cd = 0;
            }
         if (x != xh)              /* Check for change in x position */
            {
             dev += ABS(x-xh) - 1;  /* Increase deviation for jump */

             if ((x > xh) == xdir)  /*  Increase deviation by 1 if */
                 ++dev;             /*    motion in same direction */

             else xdir = 1 - xdir;  /*  Mark change in direction */
             xh = x;                /*  Save current position */
            }
         if (y != yh)              /* Check for change in y position */
            {
             dev += ABS(y-yh) - 1;
             if ((y > yh) == ydir)
                 ++dev;
             else ydir = 1 - ydir;
             yh = y;
            }
        }
    else       /*  cd < 0  ==>  last pixel scanned  */
        {
         printf("\nRegional Merit: %4.2f\n",(dev*1.0)/(VHPIX*1.0));
        }
   }

/*********************************************************************

Function:  response

Purpose:  To solicit a valid keyboard response from a user.

*********************************************************************/

char response(selstrg)
   char *selstrg;                       /*  Valid responses string */
   {
    char *ptr,ch,newline,last;

    last = '\0';
    for (;;)                            /*  Continue prompting for */
          {                             /*    valid response       */
           ch = getchar();
           for (ptr = selstrg; *ptr; ++ptr)
                if (ch == *ptr)
                    {
                     newline = getchar();
                     return(ch);
                    }
           if (ch != '\n' || last == '\n')
               {
                printf("\nIncorrect response, please try again!\n");
                fflush(stdout);
               }
           last = ch;
```

```
        }
    }

/*************************************************************************

Function: zeroarray

Purpose: To set every value of an array of integers to zero. For
         an array representing an image, this is equivalent to
         seetting the image to a solid blackground color.

*************************************************************************/

zeroarray(A,size)
    int A[],size;
    {
     int i;

     for (i = 0; i < size; ++i)
          A[i] = 0;
    }

/*************************************************************************

Function: showarray

Purpose: To display the name and contents of an array of integer
         values in a four column format.

*************************************************************************/

showarray(A,size,name)
    int A[],size;
    char *name;
    {
     int i,j,indx,rows;
     int columns = 4;

     printf("\n%s Array:\n",name);
     if (size < 1)
         printf("\n\nArray is empty.");
     else
         {
          rows = ((size-1)/columns)+1;
          for (i = 0; i < rows; i++)
               {
                printf("\n");
                for (j = 0; j < columns; j++)
                     {
                      indx = i + j * rows;
                      if (indx < size)
                          printf("    %1s[%4d] = %4d",name,indx,A[indx]);
                     }
               }
         }
     printf("\n\n");
```

```
   }

/*************************************************************************

Function: getpixel

Purpose: To return the value (0 or 1) of a pixel, selected by line
         and position, from a bit-mapped original image which is
         represented by an integer array.

*************************************************************************/

getpixel(I,lpix,ln,ps)
   int I[VH];                    /*  Original image array */
   int lpix;                     /*  Bytes/words per line */
   int ln,ps;                    /*  Line and position    */
   {
    unsigned i,w;                /*  Local processing variables  */
    unsigned b = 1;

    w = I[(ln * lpix + ps) / ISIZE];  /* Select image array byte */
    i = ps % ISIZE;                   /* Select position within  */
    return (w >> ISIZE-1-i & b);      /* Return bit/pixel value  */
   }

/*************************************************************************

Function: putpixel

Purpose: To set the value (=1) of a pixel/bit, selected by line
         and position, of a bit-mapped integer array representing
         a decompressed image.

*************************************************************************/

putpixel(D,ln,ps)
   int D[VH];                    /*  Decompressed image array */
   int ln,ps;                    /*  Line and position    */
   {
    unsigned i,j;                /*  Local processing variables */
    unsigned b = 1;

    j = (ln * HPIX + ps) / ISIZE;     /* Select array byte       */
    i = ps % ISIZE;                   /* Select position within  */
    D[j] = D[j] | b << ISIZE-1-i;     /* Set bit/pixel to 1      */
   }

/*************************************************************************/

rotate(A,B,apix,theta)
   int A[],B[VH],apix;
   float theta;
   {
    int aline,apos,bline,bpos,i,j;
    int M[3],N[2];
    int clip = 0;
```

```
    float Pi = 3.1415926;
    float R[3][2],S[2];

    theta = (theta * Pi)/180.0;
    R[0][0] =  cos(theta);
    R[0][1] = -sin(theta);
    R[1][0] =  sin(theta);
    R[1][1] =  cos(theta);
    R[2][0] =  apix/2 * (1 - cos(theta) - sin(theta));
    R[2][1] =  apix/2 * (1 + sin(theta) - cos(theta));

    zeroarray(B,VH);
    for (aline = 0; aline < apix; aline++)
         for (apos = 0; apos < apix; apos++)
              if (getpixel(A,apix,aline,apos))
                  {
                   M[0] = apos;
                   M[1] = aline;
                   M[2] = 1;
                   for (i = 0; i < 2; i++)
                        {
                         S[i] = 0.0;
                         for (j = 0; j < 3; j++)
                              S[i] = S[i] + M[j]*R[j][i];
                         N[i] = S[i] + 0.5;
                        }
                   bpos  = N[0];
                   bline = N[1];
                   if (bline>=0 && bline<apix && bpos>=0 && bpos<apix)
                        putpixel(B,bline,bpos);
                   else clip = 1;
        /*         printf("\nA[%d]  A[%d]  B[%d]  B[%d]",
                          aline,apos,bline,bpos);          */

                  }
   if (clip)
       printf("\n\n===>  Image area clipped during rotation.  <===\n\n");
   }

/*************************************************************************

Function: scale

Purpose: To scale an input or generated original image up to
         the current system preset original image size.

*************************************************************************/

scale(A,B,apix)
   int A[];                              /* Input original image       */
   int B[VH];                            /* Scaled original image      */
   int apix;                             /* Input image side in pixels */
   {
    int aline,apos,bline,bpos,n,i,j;     /* Local processing variables */

    zeroarray(B,VH);                    /* Start with background image */
```

```
    n = (HPIX/apix > 0) ? HPIX/apix : 1; /* Determine scaling factor */

    for (aline = 0; aline < apix; aline++)
         for (apos = 0; apos < apix; apos++)
              if (getpixel(A,apix,aline,apos))   /* Only scale black */
                  {                              /*   pixels         */
                   bline = n * aline;
                   bpos  = n * apos;
                   for (i = bline; i < bline+n; i++)
                        for (j = bpos; j < bpos+n; j++)
                             putpixel(B,i,j);
                  }
   }

/*************************************************************************

Function: showimage

Purpose: To display (terminal or printer) an original image up to
         a preset size.  If image is larger, it is sampled (scaled
         down) to fit.

*************************************************************************/

showimage(I)
   int I[V][H];                      /* Original image array */
   {
    int line,pos,step,i,j;           /* Local processing variables */
    unsigned b = 1;

    printf("\n\f*   T A R G E T   I M A G E   *\n\n");
    printf("\f");
    step = (VPIX/SHOWSIZE) ? VPIX/SHOWSIZE : 1; /* Set sample step: */
                                                /* equal to 1 if image */
                                               /* size <= set size     */
    for (line = 0; line < V; line++)
         {
          if (line % step == 0)        /* Step equals sampling rate */
              {                        /*   in lines and positions  */
               pos = 0;
               for (i = 0; i < H; i++)
                    for (j = ISIZE; j > 0; j--)
                         {
                          if (pos % step == 0)
                              printf("%c",
                                (I[line][i] >> j-1 & b) ? 'x' : '.');
                          ++pos;
                         }
               printf("\n");
              }
         }
   }
/*************************************************************************

Function: compare
```

```
Purpose: To compare two integer arrays, element by element, indicating
         any differences.  Used to verify that a decompressed image
         array is identical to the original uncompressed image array.

**************************************************************************/

compare(D,I)
   int D[VH],I[VH];
   {
    int i;

    for (i = 0; i < VH; i++)
          if (D[i] != I[i])
              printf("\n\n=====>  Difference at index %d  <=====\n\n",i);
   }

/*************************************************************************

Function: genimage

Purpose:  To generate a bit-mapped original image selected by
          number.

**************************************************************************/

genimage(number,I)
   int number;                          /* Selected image number */
   int I[];                             /* Receving image array  */
   {
    int i,j,k,low,high;                 /* Local processing variables */
    int *aptr;
    static int A[5][32] = {

          {   0,   0,   0,   0,   0,   0,   0,   0,  /*  Total white  */
              0,   0,   0,   0,   0,   0,   0,   0,
              0,   0,   0,   0,   0,   0,   0,   0,
              0,   0,   0,   0,   0,   0,   0,   0 },

          { 255,   0, 255,   0, 255,   0, 255,   0,  /* One quadrant  */
            255,   0, 255,   0, 255,   0, 255,   0,  /*   black       */
              0,   0,   0,   0,   0,   0,   0,   0,
              0,   0,   0,   0,   0,   0,   0,   0 },

          { 255,   0, 255,   0, 255,   0, 255,   0,  /*  Half black,  */
            255,   0, 255,   0, 255,   0, 255,   0,  /*  half white   */
            255,   0, 255,   0, 255,   0, 255,   0,
            255,   0, 255,   0, 255,   0, 255,   0 },

          {   1, 128,   1, 128,   1, 128,   1, 128,  /*  Black vertical */
              1, 128,   1, 128,   1, 128,   1, 128,  /*  line           */
              1, 128,   1, 128,   1, 128,   1, 128,
              1, 128,   1, 128,   1, 128,   1, 128 },

          {   0,   0,   0,   0,  60,  60,  60,  60,  /*  Black squares */
             60,  60,  60,  60,   0,   0,   0,   0,  /*  in quadrants  */
```

```
              0,   0,   0,   0,  60,  60,  60,  60,
             60,  60,  60,  60,   0,   0,   0,   0 } };

    if (number >= 0 && number <= 5)
        switch (number)
        {
         case 0 :     /* Image No. 0: Total white                    */
         case 1 :     /* Image No. 1: One quadrant black             */
         case 2 :     /* Image No. 2: Half black, half white         */
         case 3 :     /* Image No. 3: Black vertical line            */
         case 4 :     /* Image No. 4: Black squares in quadrants     */

                 aptr = &A[number][0];     /* Scale selected image */
                 scale(aptr,I,16);         /*   to current system  */
                 break;                    /*   standard size      */

         case 5 :     /* Image No. 5: Thick black curve              */

                 zeroarray(I,VH);
                 for (i = 0; i < VPIX; i++)
                      for (j = 0; j < HPIX; j++)
                           {
                            k = i*i + j*j;
                            high = VPIX * VPIX;
                            low  = high/4;
                            if (low <= k && k < high)
                                putpixel(I,i,j);
                           }
                 break;
         }
    else
        {
         printf("\n\nInvalid image number: %c\n",number);
         exit();
        }
    if (showarrays)
        showarray(I,VH,"I");
   }

/*************************************************************************

Function: scanimage

Purpose:  To create an original image (integer) array from an input
          stream of binary pixel values originating from a terminal
          keyboard, a stored file or a scanner.

*************************************************************************/

scanimage(I,fptr)
   int I[VH];                          /* Original image array */
   FILE *fptr;                         /* Source file name     */

   {
    int A[SCANSIZE*SCANSIZE/ISIZE];    /* Local processing variables */
    char ch;                           /*   SCANSIZE is defined size */
```

```
    int value = 0;                      /*   of an input scan line    */
    int count = 0;                      /*   ISIZE is defined size    */
    int wordct = 0;                     /*   of integer word          */
    int arraysize = SCANSIZE*SCANSIZE/ISIZE;
    int incr;

    zeroarray(A,arraysize);

printf("\n\nEnter %4d lines of %4d pixels each.\n\n",SCANSIZE,SCANSIZE);
printf("      Enter a  0  or  space for a white pixel.\n\n");
printf("      Enter a  1  for a black pixel.\n\n");
printf("      Enter a  *  to quit.\n\n");
printf("(Missing lines and unfilled line ends default to spaces.)\n\n\n");

    while((ch = getc(fptr)) != EOF && ch != '*')
          {
           if (ch == '\n')
               {
                if (count == 0)             /* At end of input line   */
                    ++wordct;               /*   complete current     */
                if (count % ISIZE != 0)     /*   word and advance     */
                    {                       /*   word count for       */
                     while (count % ISIZE != 0)  /*  unfilled portion */
                            {               /*   of scan line as      */
                             value *= 2;    /*   necessary            */
                             ++count;
                            }
                     A[wordct++] = value;
                    }
                count = value = 0;
                while (wordct % (SCANSIZE/ISIZE) != 0)
                       ++wordct;
          /*    printf("\n");     */
                if (wordct >= arraysize)    /* Discontinue accepting */
                    {                   /* input, image space full  */

                     printf("\n\nImage space full.\n");
                     break;
                    }
               }
           else                             /* Continue accepting input */
               {                            /*   (pixel) values up to   */
                if (count < SCANSIZE)       /*   size of defined scan   */
                    {                       /*   line                   */

                     incr = (ch == ' ' || ch == '0') ? 0 : 1;
            /*       printf("%ld",incr);     */
                     value = value*2 + incr;
                     ++count;
                     if (count % ISIZE == 0)  /* Collect pixel values */
                         {                    /*    into words of     */
                          A[wordct++] = value;  /*  defined size      */
                          value = 0;
                         }
                    }
               }
```

```
        }

                                        /*  End of scanned input */
    if (count % ISIZE != 0)
        {
         while (count % ISIZE != 0)     /*  Complete current word */
                {                       /*    if necessary        */
                 value *= 2;
                 ++count;
                }
         A[wordct++] = value;
        }
    scale(A,I,SCANSIZE);                /*  Scale input scanned image */
                                        /*    to defined image size   */
    if (showarrays)
        showarray(I,VH,"I");
   }
```