1987

An investigation of storage and communication codes for an electronic library

Mansour Alsulaiman
Iowa State University

Recommended Citation

Alsulaiman, Mansour, "An investigation of storage and communication codes for an electronic library " (1987). *Retrospective Theses and Dissertations*. 8608.
https://lib.dr.iastate.edu/rtd/8608

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I
Accessing the World's Information since 1938

300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

Order Number 8805038

An investigation of storage and communication codes for an electronic library

Alsulaiman, Mansour, Ph.D.

Iowa State University, 1987

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

An investigation of storage and communication codes for an electronic library

by

Mansour Alsulaiman

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Electrical Engineering
and Computer Engineering
Major: Computer Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1987

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Statement of the Problem

The library plays an important role in the academic community and the community at large. With advancement in electronic technology, it is desirable to use this technology in order to make the library more accessible to its users. It is desirable to have a library system where the user can dial up the library and access its information. The data sent should be a complete duplicate of the data in the library and not part of it. This research tries to look at one aspect of this system, namely, at the methods of compressing these data for storage and transmission.

1.2. Features and Assumptions of the Solution

The receiver in this electronic library system is assumed to originate his connection from a microcomputer. The microcomputer was chosen, instead of a dump terminal, because it provides the following necessary services to the system:

a) The receiver has a processing power which is needed to decompress the received data.

b) The receiver has storage facility. This allows the sender to send more than one page to a receiver. The receiver will work on the received data till he needs more data. This decreases the load that the sender has to manage and allows the system to service more receivers than if the receiver has to ask for the data page by page.

c) The display is of electronic form and not mechanical. Hence, the display time will be very fast. In addition

> to that, it will be negligible compared to the decompression time. Other forms such as facsimile are greatly affected by the mechanical requirements of the receiver.

In addition to the above services, the microcomputer is widely available. Hence, it is the best choice as the receiver in the electronic library system.

The microcomputer chosen for this research is the IBM PC, and its compatibles. Chapter 3 contains a description of some features of this class of microcomputers related to this thesis. The investigation carried out with this class of computers can be extended to other computers.

Since the sender is a big library system, we can assume that it is more powerful than the receiver. Hence, the compression time, that we get by simulating the compression algorithms in the microcomputer, will not be a decision factor in choosing the algorithm, unless, of course, all other factors are the same.

1.3. Thesis Organization

Chapter 2 is a review of some compression algorithms used in facsimile transmission and "Lempel and Ziv" compression algorithm. From the methods we reviewed for facsimile transmission, we chose two methods that will be investigated in Chapter 4. Chapter 3 has a description of some features of the computer this research was carried on, some implementation considerations or difficulties, and some characteristics of the data the system needs to store and/or transmit. Chapters 4, 5, 6, 7, and 8 investigate the use of some compression algorithms to compress the computer screen. These algorithms are:

a) Two methods used for compressing documents in facsimile transmission. These methods are investigated in Chapter 4. This investigation showed the need for another class of algorithms. The new algorithms should be able to detect more redundancy in the data than the two algorithms we investigated. The next chapters contain an investigation of these new algorithms.

b) Lempel, Ziv, and Welch compression algorithms is investigated in Chapter 5.

c) Variations of the Lempel and Ziv algorithm are investigated in Chapters 5, 6, and 7. These variations try to improve both the algorithm itself and the form of using it, and match these improvements to the data to be compressed.

Chapter 8 presents a general analysis of the previous methods.

Finally, Chapter 9 presents the conclusion of these investigations.

2. LITERATURE REVIEW

2.1. Review of Facsimile Transmission

An investigation of the type of data that the library possesses showed that text and graphics represent most of the data (refer to Chapter 3). Facsimile transmission is used to transmit such data; hence, it is desirable to look at the research in this field and benefit from it in solving the problem proposed in Chapter 1.

Facsimile transmission has been used since 1843 [1]. Facsimile machines consisted of electrical and mechanical systems and did not use any data compression techniques. Only in the beginning of the 1970s did some machines use a form of compression. In this review of a modern facsimile machine, we are interested only in the compression techniques it used and not in its actual structure. For an excellent source of facsimile history, development, and detailed implementation refer to [1].

The following is a review of the research in facsimile transmission. As is customary in the field, the name will be shortened to facsimile. Sometimes, it will be abbreviated to FAX in this thesis. In this review, we look at the literature in a chronological order. We will not look at all of the available literature, but we will present what we think is a representation of the available literature from the points of view of the content of the literature and the directions of the research in facsimile.

As an example of second generation facsimile machines, we look at the system described in reference [2]. The points in this paper related to this review are the following:

1) Although there were studies made on redundancy techniques, none of them was widely accepted. The reason was the unavailability, at that time, of economical methods to implement them. Advances in digital techniques and development of integrated circuits made implementing these techniques economically feasible.

2) The coding method used was to send the code of the run-length of white picture elements (pels) and send the black pels pel by pel.

3) For the high rate it was transmitting at, 50 Kbits/s, it took 20 microseconds to transmit a bit of information. This time was long enough for the recorder to guarantee sufficient exposure time for each black pel. Sending run-length of the black pels would not give enough time for the recorder to expose the black pels it should record. So, the advantage of fast transmission rate was compensated by the time increase due to sending each black pel alone. This also decreased the compression factor.

4) The paper reported a compression factor equal to 5. It also reported that other methods, that did not use this high transmission rate but used a Huffman code, had a compression factor equal to 5.7.

5) It took an average of 7 s to transmit an A4 size (8.5 x 11 in) page.

6) The paper used a variable scan rate that depended on the content of the scan line. This means when the scanner reached a black pel, it would remain 19 microseconds so the next scan would be 20 microseconds from the beginning of this scan. When it reached a white pel, it would scan normally till it reached a black pel, then it would send the run-length of the white pels.

Reference [3] gave some techniques for using the correlation between pels from line to line. It did this by ordering, in a buffer, pels or error prediction of current line based on information from current line and/or previous line. After all current line is processed, the content of the buffer is run-length coded.

The buffer filling was tried using the following methods:

1) Each pel in line i+1 is put to the left (right) of the buffer if the same pel in the previous line is white (black).

2) Each pel in line i+1 is predicted to be the same as the pel in line i. The error in prediction is ordered as in method 1, i.e., if the same pel in line i is white (black), the error prediction is put to the left (right) of the buffer.

3) Each pel in line i+1 is predicted depending on its state. The state of a pel was defined as the three pels in line i nearest to the pel plus the pel to its left in line i+1. The prediction error is put to the left (right) if the prediction is good (bad). The ordered buffer is then sent as run-length codes. A prediction is classified as a good one if its probability is bigger than a threshold (0.8); otherwise, it is a bad one (note that a probability is defined to be at least 0.5).

These methods gave a compression factor that is 30-50% better than the one of a one dimensional run-length coding. It was also shown to be 10-18% better than the compression factor of another ordering technique suggested by Preuss (refer to discussion of reference [4]).

Reference [5] is a continuation of the work in [3] done by the same authors. It used the ordering technique that depends on the state of the pel as described in the discussion of [3]. It had the following enhancements:

1) It used 7 previous pels instead of 4.

2) The threshold of a good prediction was raised from 0.8 to 0.9.

3) The statistics of the prediction were averaged from the 8 standard documents suggested by the International Telegraph and Telephone Consultative Committee, known as CCITT.

4) The first sequence of 00...01 in the buffer would not be sent.

5) Each line was ordered from either left-to-right (forward) or right-to-left (reverse) depending on which direction gave better result, i.e., needed less bits.

The method was tried on all the 8 CCITT documents and an average of 41% decrease in the transmission time compared to the transmission time obtained using the modified Huffman code was reported.

Reference [6] is an invited paper by Huang which reviewed some of the coding methods available at its time. The paper gave three heuristic concepts used in facsimile coding. They are the following:

1) Skipping white: Only the black elements will be sent and the rest of the document is assumed to be white.

2) Transmitting only boundary points: It is perhaps fair to say that the majority of the current efficient coding schemes are based directly or indirectly on this concept. Examples of how this is done are sending the address of the boundary points, counter tracking these points, and approximating boundaries by piecewise linear or polynomial curves. Later, the paper gave more practical examples.

3) Pattern recognition.

Some mathematical models were given, corresponding entropies were derived, and numerical examples of their values were given. The white block skipping scheme was shown in one and two dimensions. It was also shown how to make it adaptive. Run-length coding was discussed and a mathematical model and experimental results were given. Two forms of predictive differential quantization were also given. Preuss code was presented as another form of an extension of run-length coding. Besides, the paper noted the following general trends:

1) For low resolution, 100 pels per inch (ppi), one dimensional coding techniques were usually preferred because of the ease of implementation and because they gave compression factor comparable to the one of the two dimensional coding.

2) For high resolution, greater than 200 ppi, two dimensional coding techniques may give considerably higher compression factor and be preferred in spite of their implementation complexity.

Reference [4] was an attempt to compare some of the codes submitted to CCITT for standardization of group 3 facsimile machines. It looked at some one and two dimensional coding techniques.

The one dimensional techniques were all run-length coding techniques. They differed according to the code assigned for the runs. One of these techniques that used the Modified Huffman (MH) code would be the one dimensional standard recommended by CCITT.

The two dimensional codes were:

1) The Kalle-Infotec code: It works on a pair of consecutive lines that are segmented into black and white runs. The runs for both lines together are coded with an adaptive run-length code which changes its word length between 2 and 8 bits according to the local statistics of the document.

2) The Kokusai Denshin Denwa code: It is similar in principal to the EDIC code that we will discuss later.

3) Preuss code: Sometimes, it is referred to as the TUH (Technical University of Hannover). In this code, each pel is predicted from the nearest 3 pels in the previous line and the pel to its left in the current line. These 4 pels form a state for that pel. For each pel, the code uses its state to predict its value. A value of 0 or 1 is inserted in its place in the current line depending on the prediction error. For each state (16 states) the run length between its prediction errors is coded using a truncated Huffman table. Each state has its own table which is constructed from statistics of type written text.

Among the two dimensional techniques, the TUH had the biggest compression factor specially for documents filled with a lot of text. The three one dimensional methods had almost the same compression factor, but MH had the biggest one.

Two dimensional techniques yielded a considerable gain (average = 16%) over one dimensional techniques only for high resolution. For low resolution, the difference between one dimensional and two dimensional techniques was minimal specially for text documents.

Reference [7] discussed the features and design of a display processor that can output both text and graphics to a display at the same time. The processor consisted of two data paths that operated in parallel. The data from both paths were logically ORed together and output to the display.

The first path was the character generator that changed the text information from code (ASCII code and/or control code) to a bit map representation of the characters. The text format was variable so different sizes could be output. This meant that text could have subscript, superscript, invert, and other formats. The second path was the FAX generator that took compressed data of an image, decompressed it, and then sent it to the display so it could be superimposed on the output of character generator.

The display resolution was 120 pels/in horizontally and 96 lines/in vertically. The images to be superimposed were assumed to have large empty areas (i.e., white color) and tended to have large numbers of horizontal and vertical lines. The resolution of the scanner was the same as the one of the screen. The main goals were to have a fast method of decompression that could decode the compressed data without using any image buffer to store the complete picture, and the decoding method should be simple to be implemented. This was done by decoding the screen part by part from top to bottom then restarting this process again. The compression/decompression method used was a combination of block coding (refer to discussion of [8] below), simple run-length coding, and very simple prediction. Since this method was not designed to give an optimum compression factor, this review will not discuss it furthermore.

Reference [9] described the Edge Difference Coding (EDIC) technique. This technique looks at the current and previous line from left to right looking for the next two color changing pels, and then defines a state out of the following three states:

1) State S1: One transition pel is in the current line and the other one is in the previous line.
2) State S2: Both transition pels occur in the preceding line.
3) State S3: Both transition pels occur in the current line.

The states are then coded as follows:

1) State S1: A code for the distance between the two pels would be sent.
2) State S2: A code to signal that this state had happened would be sent.
3) State S3: For each of the two transition pels, a code of the run length that ended before it would be sent.

Reference [10] is a short review of facsimile development and its current state from the point of view of speed, technologies used, and specific machines. It covers both analog and digital facsimile. One example of analog facsimile decreased transmission time by bandwidth reduction. Another analog facsimile decreased transmission time by scanning faster, on the sender and receiver, over white areas. No redundancy reduction algorithm was presented.

Reference [11] discusses a system that uses a method called Combined Symbol Matching (CSM) for facsimile compression. The system works in the following two stages:

1) Symbol Matching: In this stage, the system tries to find the basic symbols, e.g., alphanumeric characters, of the document. It scans for symbols till it finds one. Then, it will compare the found symbol with the library of symbols the system encountered before. The comparison uses some symbol features as a preliminary screening before it performs the bit map comparison. If a match is found, the symbol number in the library will be sent with its relative location from the previous symbol. If no match is found, the symbol with its features and bit map will be added to the library and its bit map, width, height, and location will be sent to the receiver. Any symbol that is sent is replaced by white space. After all symbols are processed, the next step starts.

2) Residue Coding: By residue, it is meant the document without the symbols sent in stage 1. This residue is coded by a two dimensional run-length ccding and sent to the receiver.

The compression factor found by this method for compressing the CCITT documents (resolution was 200 x 200 lines/in = 8 x 8 pels/mm) is a 2 to 3 times READ's (Relative Element Address Designate) compression factor for document 5 and about the same for document 2. A pattern recognition was tried and resulted in compression factor greater than 250 for compressing a business letter.

We would like to make note of the following points:

1) There were some overhead bits sent whether symbols were matched or not. No matching has higher overhead.

2) The paper allowed for small error in matching the symbols. When it tried exact matching, a decrease of 50% in the compression factor was reported.

3) The code was asynchronous. For each matched symbol, some overhead (e.g., shift up or down, distance to previous block) was needed to be sent, whereas for each non-matched symbol its size and its distance to the previous symbol were sent. For each line, the location of the first pel on the line and a flag to indicate if there was a symbol or not would be sent. These overheads complicate the coding and decrease the compression factor.

Reference [12] is an invited paper that gives an overview of digital facsimile coding techniques in Japan. The author classified the two dimensional information preserving codes into line by line coding and simultaneous coding of n lines. For simultaneous coding, he gave the following three examples:

1) Mode Run Length Coding: It examines n lines at the same time. For each horizontal pel location, a state is defined depending upon the corresponding pels in the n lines. The code sent is the run-length code of the state with a variable length code for state to state transition.

2) Coding by Zig-Zag Scanning: The pels are read in a zig-zag way (i.e., we jump from reading a pel in line i to reading another pel in line i+1, then we go back and read a new pel in line i, and so forth). A simple run length coding of the encountered bits does not work well. One technique to solve this problem is to predict the pel based on the three pels read before it. Then, the runs of correct and erroneous predictions are run length coded by a suitable code for each of them.

3) Cascade Division Coding: This is almost similar to the block coding in [8].

The author mentioned that recent trend had recognized line by line coding as the most favorable approach for two dimensional coding. He then gave the following examples of line by line coding:

1) Two Dimensional Prediction Coding: It is one of the earliest proposals. Other coding methods such as Preuss' or the one in [4] had this method as a step within many steps. So, we will not discuss it.

2) Relative Address Coding (RAC): It has the same general principals of PDQ and EDIC. The author suggested that although PDQ was known first, RAC was one of the landmarks in the history of facsimile. He attributed this to the fact that PDQ was not described as a practical coding scheme and no comparison with simultaneous coding scheme was available. But RAC was the first method to present the fact that line-by-line coding could, indeed, give better compression factor than simultaneous coding. It works by sending the code that specifies the positions of the changing elements in each line. The position of each changing element is sent by sending the code of the shortest following two distances: the distance between the current changing element and the previous one in the same line, or the distance between the current element and the nearest one in the line before it.

3) Edge Difference Coding (EDIC): It was explained in our discussion of [9].

4) Coding by Rearranging Picture Elements: This is divided into microscopic and macroscopic rearrangements. The method by Mounts et al. [5] is similar to but more advanced than the microscopic method the author reviewed. The macroscopic rearrangement is done by finding the size of the characters and then arranging the characters of each line at its left. The arranged image is then coded by microscopic coding.

5) Coding by Classified Pel (CP) Station: The basic idea is similar to Preuss' method; hence, we will not discuss it.

6) Relative Element Address Designate (READ) Coding: It combines features of RAC and EDIC. A modification of it,

called Modified READ (MREAD), was accepted by CCITT as the standard code for two dimensional coding (refer to discussion of [13]).

Reference [13] describes the CCITT standard for one and two dimensional coding of documents for facsimile transmission. This standard has been drafted by Study Group XIV of CCITT as recommendation T.4 for what is called Group 3 facsimile machines. The elements of this standard that are important to us are the following:

1) Resolution: Each scan line on an A4 size document is divided into 1728 pels. The normal vertical resolution is 3.85 lines/mm. A higher vertical resolution of 7.7 lines/mm is available as an option.

2) Timing: Due to mechanical limitation of some machines (specially in the recorder part), a minimum transmission time is assured for each line so that the sender and the receiver can be synchronized together.

3) The one dimensional code: It was decided to use a run-length coding technique. Huffman coding was chosen because of its good compression factor. The paper reported that an experiment showed that the error recovery of Huffman code was comparable to other codes. Instead of coding the length from 0 to 1728, it was decided to limit the size of the table by using make-up words. Hence, this table was named the modified Huffman table.

4) The two dimensional code: Several proposals were submitted. The committee chose READ (suggested by Japan) and added some modifications to it. Hence, the code is called the modified READ (MREAD). The committee found the compression factor of READ to be the same as the one of other proposals. But READ was chosen because it has been implemented in a large number of commercial machines (Japan depends a lot on facsimile, refer to [12]).

Then the paper also discussed the error recovery of both the one and two dimensional standards. This error recovery will not be discussed in this review. It also gave some simulation results of one and two dimensional standards applied to the CCITT documents.

Reference [14] derived the entropy of RAC method, a scheme based on non-Markovian grammar. It gave numerical examples to prove the correctness of this derivation and the wrongfulness of another method, presented by other authors, which used 2nd order Markovian model. The error in the numerical values was an order of magnitude.

Reference [15] is a modification of Preuss' method. In this method, after predicting the new line from the old one and finding the prediction errors for each state, the length to be coded is the length from the state first correct prediction, in a sequence consisting of the same states, to the current state error in this sequence.

Reference [8] has many good points besides its coding method. So, we will present its steps in the following:

1) It used a set of masks to remove notches and pinholes from the scanner output. The notches are mostly caused by the presence of imperfections in the scanning process. Removing these nothces improves the coding efficiency and, to a certain extent, improves image quality.

2) For every single black pel between two or more whites, another one is inserted before it. This is necessary so that no loss of information will occur after the next step.

3) The image is subsampled in horizontal and vertical directions by taking every other pel in these two directions. Hence, resolution is reduced by a factor of 4.

4) The picture is divided into blocks of certain size called Initial Picture Block (IPB). If the IPB is not either all white or all black it is divided into 4 subpictures blocks (SPB) and a code of the division is sent. Each SPB is tested to check if it is all white or all black, if no further division is made. When an all white or an all black SPB is found, a code for it is sent. The division con-

tinues (if no all white or all black is found) till an SPB of size 4, called basic picture block (BPB), is reached. The BPBs are Huffman coded according to the position of the black pels among its 4 pels.

5) The received data are used to construct the subsampled data which are interpolated to get the original data. Three methods of interpolation were used, namely, bilinear, replication, and B-spline. Subjective tests were made and led to the conclusion that bilinear was almost the best of the three methods. An average of 20% decrease in quality was noticed in these tests.

6) Due to the interpolation, some extra points might be generated. Some restoration matrices were used with two of the interpolation methods to get rid of these points.

The CCITT documents were scanned and compressed. The compression factors were compared with the ones of the MH (in original and subsampled form) code and found to be better. But, if we compare the ratio of its compression factor to the one of the MH subsampled, it is found to be almost the same as between MREAD and MH (neither MREAD nor MH in this case is subsampled). So, no big gain in compression factor was due to the coding method itself, except maybe for document 2. The following three IPB sizes were used: 8 x 8, 16 x 16, and 32 x 32 pels. Bigger sizes were not used and the paper suggested that no further substantial increase in compression factor could be achieved in this way. The compression factor generally increased with the size increase of IPB. This is maybe due to the extra overhead bits needed in coding smaller IPB sizes.

Reference [16] is an example of progressive image transmission technique. It transmits defined pieces of the image till the whole image is transmitted. The benefit is that most of the details can be seen faster and we may stop at a stage before sending the whole data and still get a good image. It transmits in 7 stages as follows:

1) Every line numbered a multiple of 16 is transmitted with 1/4th of the horizontal resolution.

2) Another line out of 16 is transmitted at the same horizontal resolution. Each of these lines will be in the middle of two previously transmitted lines (i.e., in stage 1 we transmitted lines 1, 16, 32,... and in stage 2 we transmitted lines 8, 24, 40,...).

3) One of 8 lines is transmitted. These lines (numbered 4, 12, 20,...) are in the middle of lines transmitted in stage 1 and stage 2. So, after stage 3, every fourth line is received at 1/4th of the horizontal resolution.

4) The horizontal resolution of transmitted lines is doubled. So, every fourth line is received at half resolution.

5) One out of 4 lines (e.g., lines 2, 6, 10, 14,...) is transmitted at half resolution.

6) The horizontal resolution for previously transmitted lines is doubled. So, at the end of this stage, all lines are with full resolution. These lines are the even lines.

7) The odd numbered lines are transmitted at full horizontal resolution.

The lines sent at each stage are coded using CCITT code (both one dimensional and two dimensional). Note, that for half horizontal resolution, each element is replaced by two pels on the screen.

The paper suggested that stage 5 could be considered as the last stage for screen display since it requires 864 pels/line and 1188 lines/page which is the resolution limit of high resolution monitors.

Reference [17] is another progressive transmission technique. It has four stages. The image is sampled at 1/4th of both the horizontal and vertical resolutions. These samples are coded by one dimensional code and the codes are sent to the receiver that interpolates the missing pels. In the next three stages, run length codes of the prediction errors of the remaining pels are transmitted. The prediction used previously transmitted pels as the reference for prediction.

Reference [18] presented an experimental system of facsimile communication using packet switched data network (PDSN). Facsimile is usually sent by telephone over public switched telephone network (PSTN). The paper gave the communication protocols and the needed processors for the experimental system. It also used the facsimile standard of group 3 machines.

Reference [19] described features of an apparatus for fast documents transmission over a 1.536 Mbits/s satellite link without redundacy reduction. It presented new techniques for recording a system and its control procedure.

Reference [20] presented error sensitivity of both the one and two dimensional facsimile coding standards. As expected, it was found that two dimensional coding was more affected by errors than the one dimensional coding. The paper discussed ways to stop the error effect from spreading throughout the page.

Reference [21] described a facsimile compression system that uses a symbol matching technique. It used the same principal as in [11] with some modifications and presented more details of both the symbol matching and the features extraction. It had two more features to be extracted than the features in [11]. It reported that these two features offered higher degree of symbol identification. The paper also showed that some signal modification techniques, applied before the two dimensional coding, resulted in a typical 14% improvement over regular two dimensional coding.

Reference [22] used a symbol matching technique similar to the one in [11] and [21]. It was more enhanced, more optimized, and did not have residue coding. The main advantages of this new technique are the following:

1) It matches not only symbols but also nonsymbol patterns. A nonsymbol pattern was defined as a pattern of certain size and window, and that has a black pel in it which is connected to other black pels outside the pattern. An example of this is parts of vertical and horizontal lines. the symbol is defined as a pattern that has connected black pels, is totally surrounded by white pels, and fits inside a window. This allows the method to efficiently code graphics. So, all black data are coded and no residue is left. This, of course, implies a white background.

2) The symbols in a line are stored and arranged in a buffer before sending them to the receiver. This resulted in efficient coding. Example of this efficiency is that it arranges the same symbols after each other and does the following: the code of a repeated symbol (i.e., its library number) is sent first for its first occurrence. Then, for the coming consecutive occurrences of this symbol, we send a shorter code (3 bits) that signals the receiver that the library number is the same as before.

3) It used a better criterion for symbol matching.

4) The bit map was compressed by the CCITT two dimensional code before sending it.

5) The coding of the data was more optimized and used variable length code for control information.

6) The library management was better and the library size was bigger.

7) The compression factor ratio to the one of CCITT two dimensional code was often doubled and it reached 4.5. Compared to CSM, it was 20-80% bigger.

8) For CSM and this method, the compression factor doubled between two versions of the same document that differed in resolution.

9) By using mixed custom and programmed logic, it was able to send a document in one to two seconds at a 64 kbits/s rate.

Note that the paper reported wrong matches (e.g., between O and 0, i and 1).

Reference [23] describes algorithms used in the design of Image View Facility (IVF), a system/370 based software that permits the display and fast manipulation of binary images. This software allows images to be rotated, scaled (so it can be displayed at different resolutions), and compressed. The compression algorithm is a slight modification to MREAD. It modifies MREAD by dropping the end of the line sequence, not inserting any fill bits, and using an end of the document sequence. The paper reported an increase of the compression factor by 15 to 35% when these modifications were added to the case of not using them. The images to be compressed had the same horizontal resolution as CCITT standard, but the vertical resolution was slightly different (1100 and 2200 lines/page for low and high resolution, respectively). The decompression time was found to be 3 to 10 times faster than the authors anticipated.

From the above review, we come to the following conclusions:

1) Line-by-line techniques are the best among the techniques that do not have any symbol matching capability. Practically, there is no difference between the line-by-line techniques, so MREAD can be chosen because it is the standard.

2) Line-by-line techniques, even though called two dimensional coding, are a limited form of two dimensional coding because first, these methods use no memory to remember the content of more than one reference line. Second, the coding line uses only a small part of the information available in the reference line.

2.2. Review of the Lempel and Ziv Algorithm

The investigation in Chapter 4 will show that the compression methods used in facsimile, except those that use pattern recognition or symbol matching techniques have two problems. First, they do not give the same compression factor they give in facsimile machines. Second, they are limited in the amount of redundancy they can recognize. Therefore, a new type of algorithms should be investigated. The universal coding algorithms are such algorithms. From these universal coding algorithms, we chose the Lempel and Ziv algorithm which we will review in the rest of this chapter. For a review of universal coding, refer to [24]-[31].

The Lempel and Ziv method for data compression looks at the data as a string of symbols. This string is a collection of smaller strings (substrings) of symbols (substrings may overlap). These substrings are generated from previously encountered substrings and some symbols. While this method scans the string, it builds a table of these substrings and sends a code of the current substring. By finding the best substrings to represent the original string, we get a total size of the sent codes that is smaller than the size of the original string; hence, the data are compressed.

In the following review, we will look at papers that dealt with the Lempel and Ziv method, including papers by the authors themselves. For the sake of following the method development, we look at the papers in their chronological order.

The following abbreviations will be used:

LZ = Lempel and Ziv

LZ method (or theorem) = The Lempel and Ziv method (or theorem).

LZW method (or theorem) = The Lempel, Ziv, and Welch method (or theorem). It is a modification and clearer representation of Lempel and Ziv's method done by Welch. This method is the one we will be using later.

$\lfloor x \rfloor$ = The smallest integer bigger than x.

In [32], Ziv proposed two forms of the probability of the block coding error. He then proved the existence of a universal constant code for which the error probability (using both forms) goes to zero as the code length goes to infinity.

An algorithm for coding was given in [32]. It works as follows:

- The message is divided into blocks of n letters each.
- Each block is divided into n/k vectors (k-grams).

- Each vector (gram) is translated into a code which is a $(k \lfloor \log_2 L \rfloor)$ vector, where L is the size of the source alphabet.
- The code word of a block consists of nR binary letters (bits), where R is the coding rate.
- The code is divided into two parts:
    - a) a list of the distinct vectors in the n letters.
    - b) a sequence of codes for the (n/k) vectors where each code is an address for a word in the list of distinct words in part a above.

It was shown that the probability of an encoding error can be made small for output rates which are not larger than those of the optimal codes that do depend on the statistics of the source.

In [33], Lempel and Ziv looked at the complexity of finite sequences. They proposed linking the complexity of sequences to a gradual build up of new patterns along each sequence from a finite alphabet. Works before this tried to define the complexity of the sequence by linking it to an algorithm by which the sequence is supposed to be generated. This definition of the complexity is not offered as a new absolute measure of complexity, which the authors believe nonexistent. Rather, it evaluates the complexity from the point of view of a simple learning machine which, as it scans an n-digit sequence $(S = s_1\ s_2\ s_3\ \ldots . s_n)$ from the left to the right, adds a new word to its memory every time it discovers a substring of consecutive digits not previously encountered. The size of the vocabulary and the rate at which new words are encountered along S serve as basic ingredients in the proposed complexity evaluation.

The proposed measure is defined and put to test against a well-established test case, namely, the de Bruijn sequences. Under this measure, it was shown that most sequences are complex. However, it was also shown that this measure was not very weak, by showing that it discarded ergodic sources with normalized entropy less than one.

The paper laid down some definitions of sequences build up and sequences parsing. The "reproduction" and the "production" of a sequence from its parts were defined.

The complexity of S was defined as follows. Any nonnull sequence S can be parsed into its history as in $H(S) = S(1, h_1)\ S(h_1 + 1, h_2) \ldots S(h_{m-1} + 1, h_m)$. These m strings are called the components of H(S). A component $H_i(S)$ and the corresponding production step, $S(1, h_{i-1}) \Longrightarrow S(1, h_i)$ are called exhaustive if $S(1, h_{i-1}) \not\longrightarrow S(1, h_i)$, where $\Longrightarrow$, $\longrightarrow$, and $\not\longrightarrow$ mean produce, reproduce, and do not reproduce, respectively. A history is called exhaustive if all of its components except the last one, are exhaustive. Every nonnull sequence has an exhaustive history.

Let's now define the following terms:

$c_H(S)$ = The number of components in a history H(S) of S.

$c(S)$ = The proposed measure of complexity of the sequence S

= $\min \{c_H(S)\}$.

$c_E(S)$ = The number of components in the exhaustive history of S.

It was proved that $c(S) = c_E(S)$. An upper bound for $c(S)$ was given in terms of n and α, where n is the code length and α the size of the input alphabet. It was shown that for almost all strings S, $c(S)$ was close to this upper bound.

The main idea from this paper that will be used in the following papers is the way strings can be built and their proposed complexity measure.

Using the concept of string copying procedures introduced in [33] for building sequences from the parsing of its individual substrings with minimum number of steps, [34] introduced an algorithm for compressing the sequence without prior knowledge of its statistics. The effect of source statistics on the code manifests in building the string from previously encountered strings.

The encoding algorithm proposed by [34] can be explained as follows:

- Let A be a finite alphabet of α symbols and S a sequence of letters from the alphabet (A $S = s_1\ s_2 \ldots\ldots. s_{l(s)}$, where $l(s)$ = length of S).
- $S(i, j) = s_i\ s_{i+1} \ldots. s_j$.
- For each j, such that $0 \leq j \leq \ell(s)$, $S(1, j)$ is called a prefix of S; $S(1, j)$ is a proper prefix of x if $j \leq \ell(s)$.
- For $S(1, j)$ and i, where $i \leq j$, let $L(i)$ denote the largest nonnegative ℓ, where $\ell \leq \ell(s) - j$, such that $S(i, i+\ell-1) = S(j+1, j+\ell)$. p is the position within $S(1, j)$ for which

$L(p) = \max\ \{\ell(i)\}$; maximization is over i, where i is in the range [1, j].

- The substring $S(j+1, \ell+L(p))$ of S is called the reproducible extension of S(1, j) into S and the integer p is called the pointer of the reproduction. So, although S(1, j) may reproduce, i.e., by copying, different extensions bigger than S(1, j), we choose the longest extension to be the reproducible one.
- The encoding is done by parsing S into $S = s_1\ s_2\ s_3\ldots$, where s_2 is the reproducible extension of s_1 into S and s_3 the reproducible extension of $s_1\ s_2$ into S, and so on. Each s_i is assigned a code c_i (c_i has a fixed length).
- To get a bounded delay encoding, a buffer of finite length n is used to hold the last encountered symbols. The parsing is modified by limiting $\ell(s_i)$ to a maximum value of L_s. The parsing is done now by finding the reproducible extension of $B(n-L_s)$ into B, where B is the buffer content.

The encoding proceeds as follows:

1) Initialize the buffer to $(n-L_s)$ zeros (the left side of the buffer) followed by the first L_s symbols of the input string S (reading S from left to right). This content of B is B_1.
2) Having determined B_i, look for the reproducible extension E of $B_i(1, n-L_s)$ into $B_i(1, n-1)$. From E, get s_i = E.s

where s is the symbol next to E in B_i. For B_i, let $\ell_i = \ell(E) + 1$.

3) Let p_i be the reproduction pointer used to determine s_i, then the code word c_i for s_i is given by $c_i = c_{i1}\ c_{i2}\ c_{i3}$ where:

   $c_{i1} = (p_i - 1)$, so $\ell(c_{i1}) = \lfloor \log_2 (n-L) \rfloor$.

   $c_{i2} = (\ell_i - 1)$, so $\ell(c_{i2}) = \lfloor \log_2 L_s \rfloor$.

   c_{i1} and c_{i2} are in radix α representation.

   c_{i3} = last symbol of s_i (i.e., $c_{i3} = B_i(n-L_s + \ell_i)$).

   Send out the code c_i.

4) Shift (to the left) out of the buffer the symbols occupying the first left ℓ_i positions while feeding in the next ℓ_i symbols from the source.

5) Go to step 2 and continue till all the string S is encoded.

Decoding is done by reversing the encoding process, it works as follows:

1) Use a buffer of length $(n-L_s)$, initializing it to zeros. This is B_1.
2) From c_{i1} and c_{i2} determine p_i and ℓ_i.
3) Store the content of $B_i(p_i)$.
4) Shift to the left B_i one time. Put the stored $B_i(p_i)$ in $B_i(n-L_s)$.
5) Continue the storing, the shifting, and the filling for $\ell_i - 1$ times.

6) Shift B_i to the left one more time and then fill $B_i(n-L_s)$ with the symbol s which comes from c_{i3}.
7) S_i is now in $B_i(n-L_s-\ell_i, n-L_s)$ which is the ℓ_i far right positions of B_i.
8) Go to step 2 and continue till all the c_i's are decoded.

Reference [34] derived bounds for block-to-variable and variable-to-block coding designed to match a specific source. Then, it derived the bound for this universal coding and showed that it uniformly approached the lower bounds for the two coding methods.

Reference [35] defined the finite state encoder and decoder and restricted the discussion to this class of machines. This machine has a memory and encoder (or decoder) delay time. Two examples of this class were given, one of them was a block encoder. The block encoder was the one that was used in the rest of the paper.

For faithful coding, under constant coding and decoding rate, the paper defined the quantity h(u) and showed that it played a role analogous to that of the entropy, although no statistical information was used to get h(u). The analogy came from finding that, using the coding method introduced in [32], the coder input did not equal the decoder output if $h(u) > \log_2 \beta$, where β is the size of the output alphabet. h(u) is defined as a measure of the complexity of the sequence:

$h(u) = \lim_{\ell \to \infty} h_\ell(u)$, where $h_\ell(u)$ is given by

$\log_2 \ell h_\ell(u)$ = number of distinct ℓ vectors in an infinitive sequence u.

From h(u), the source complexity H(u) was derived. It was also shown that the entropy of a source equaled its complexity, H(u), for an ergodic source, and the expected value of the complexity for a stationary source.

It was also shown that a normalized version of the Lempel-Ziv complexity, defined in [33], was a lower bound on H(u).

Reference [36] took the concept of universal coding introduced in [34] and applied it to variable rate coding. The way it parses a string is the same, but the way it codes individual parameters is different. The paper also defined the compression ratio of a finite state encoder in terms of the block length, the code length, and the size of the source symbols. From the compression ratio, the minimax $\rho(X)$ is defined as the finite state compressibility of a sequence x (as block length goes to infinity and number of states goes to infinity).

Reference [36] also showed that $\rho(x)$ had a lower bound in terms of the normalized Lempel-Ziv complexity (defined in [33]). $\rho(x)$ also has a role analogous to that of the entropy (as did the quantity H(.) defined in [35]).

Reference [37] showed that there existed an asymptotically optimal universal coding scheme (the encoder is assumed to be an information lossless finite state encoder, whcih is defined in the paper) under which the compression ratio of a string x tended in the limit to the compressibility $\rho(x)$ for every string x.

A direct application of LZ method, as presented in [33], needs calculations of $O(n^2)$, where n is the string length. To overcome this problem, [37] used an algorithm of tree construction due to McCreight. The parsing of the string is done by building a compact tree which is linear in n. Then, McCreight algorithm makes it possible to construct this tree in a time linear in n, i.e. O(n).

Using this method and a universal presentation of integers yielded a universal linear variable-to-variable encoding scheme. The compression ratio of this scheme was shown to be optimal for ergodic sources as the length of the input string goes to infinity.

Reference [38] looked at the LZ algorithm as an example of data compression via textual substitutions or macro coding. It classified macro coding into two classes, namely, external and internal macro schemes. Each class is divided into subclasses. LZ method falls under the subclass called original pointer macro coding in the internal macro scheme class (an original pointer is defined as a pointer that points to a substring of the original string).

Reference [38] then related the performance of the LZ method to other classes showing that the worst case performance of LZ did not compare favorably with other schemes. It also mentioned that LZ was asymptotically optimal for ergodic sources as the source length tended to infinity, but for individual finite strings it could be far from optimal.

Reference [39] showed that for parsing strings, the greedy dissectors, such as LZ, were optimal for some classes of strings but not for others.

Reference [40] showed that LZ method could be represented by an incomplete parsing tree. It then showed that the working of LZ could be explained by an equivalent symbolwise model. This representation gave more insight on the work of LZ and why it compresses the strings.

In [41], Welch gave a modification of LZ method and showed more clearly how to use it. We delay discussing it to a later chapter to avoid repetition.

Reference [42] looked at three compression schemes, namely, LZ method, arithmetic coding, and Huffman coding. It gave some bounds for each of them and did some simulation to compare them. The simulation gave better results than the bounds did. It also gave the following interesting results:

1) For the data that occupy a small size memory (less than 1KB), it is recommended to use the arithmetic coding. For the data that occupy a medium size memory (few KB), the Huffman code is the best. For the data that occupy a big size memory (tens of KB), the LZ coding (which it called universal coding) is better than the other two.

2) The cross point between the algorithms, as memory varies, depends on the source entropy. For instance, if memory equals 1KB the cross point between the arithmetic and the Huffman coding is at entropy equal to 0.19. This means that for a data of size 1KB, Huffman coding is better for entropies bigger than 0.19.

Reference [43] gave a modified LZ coding which finds out the basic building blocks (words or sentences) of the language and synchronizes itself on these blocks. It achieves this by searching for a new string match then letting this match be the extension of the string method in the last previous search. The memory requirement is the same as in the LZW algorithm but it requires complex programming to solve some special cases.

A simulation result showed that this algorithm compression factor was slightly less than the one of LZW for an English text and a Fortran source code and bigger for a pseudo random sequence. An interesting note, which [43] did not mention, is that this algorithm gave better results as the entropy increases (the best result was for the pseudo random sequence). Using a variable coding for the output improved the compression slightly (6%).

Reference [43] showed that for the basic LZ the binary representation is better than the one byte representation because the new symbol is smaller in the first case (one bit vs. 8 bits). This problem can be solved by including the new symbol as first symbol of new string (as in LZW).

It also showed that choosing the basic building blocks (i.e., 4, 8, 16 bits) as the symbols was better than the others (e.g., 3 bit symbols).

In [44], Lempel and Ziv tried to extend their universal code to picture compression. They did this by using one of the color filling algorithms to scan within subblocks of the picture. The intuition about this is that this way of scanning the picture will produce for each block a string that is more suitable to the compression than the string of a normal scan. The order of moving from a subblock to another also tries to exploit this more by avoiding the move to a subblock that is far in the picture but next in order in a normal scan. It does this by moving forward then backward (or upward then downward) instead of moving forward from one end to another then retracing to a lower block to start a new block.

Our intuition is that this method may not be suitable to our specific goal because of the following reasons:

1) It works on square pictures; but our way of dividing the picture into blocks according to their class of content, will mostly produce rectangular blocks instead of squares.

2) It is suitable for blocks of colors, but for graphics or complex colors we think it will not work much better than normal scanning will.

Due to time limitations, this method will not be checked.

3. CREATION OF THE IMAGE DATA BASE

3.1. Classification of the Library Informational Material

A survey was done to get an idea about the type of information contained in typical library materials. The subject of this survey was selected magazines that are thought to be representative of the other magazines in the library. The magazines were chosen because they will be more used in the electronic library than other materials like books. Besides that, magazines contain more colors and photos. Hence, they occupy more memory in storage and take longer transmission time.

The results of the survey are shown in Table 3.1. Under each class of data in this table, column "b" represents the percentage of the size of this class to the size of the whole document. For all classes except "text" and "space" classes, column "a" is the percentage of pages containing that class to the total pages of the whole document. Column "a" in "text" is the percentage of pages containing text only to the pages of the whole document. It is meaningless to have a column "a" in the class "space" because all pages contain some amount of space.

The average of each column in Table 3.1 was calculated. It showed that text represented 57% of the data and space represented 13.5% of the data. Black and white photos, colored photos, and graphs classes represented no more than 10% each. The percentage of pages containing only textual data represented an average of 33% of the total pages in each document.

Table 3.1. Results of the library data survey

Periodical	% text a	% text b	% space b	% b/w photos a	% b/w photos b
Polymer Science	30.00	57.00	20.00	1.20	0.30
Bios	62.00	66.00	9.20	18.80	11.30
The American Biology Teacher	29.00	62.00	11.00	30.80	7.50
Mechanical Engineering	14.30	44.30	10.70	30.00	9.80
Business Review	19.60	63.80	17.30	27.00	7.60
Welding Journal	3.70	40.40	9.70	38.20	11.30
Ergonomics	55.80	70.70	16.20	0.96	0.50
Aerospace	0.00	40.70	13.50	15.00	0.70
Sight and Sound	1.30	57.20	5.70	80.00	24.90
Nebraska Farmer	0.00	28.50	13.70	57.30	17.80
Political Methodology	79.00	64.00	26.00	0.00	0.00
National Journal	29.00	68.00	14.00	54.00	14.00
Higher Education	79.00	77.00	8.90	0.00	0.00
International Journal of Computer And Information Science	59.00	65.00	14.00	0.00	0.00
AVERAGE	32.98	57.47	13.56	25.23	7.55

% color photos		% graphs		% tables		Sum
a	b	a	b	a	b	b
2.50	2.50	61.00	16.60	5.80	0.66	97.06
5.00	5.00	12.50	6.70	3.80	1.00	99.20
11.80	7.90	57.00	10.00	1.50	0.30	98.70
41.40	26.00	35.00	8.50	0.00	0.00	99.30
2.20	2.20	43.00	6.00	3.30	0.70	97.60
39.70	23.80	69.90	12.30	6.60	1.60	99.10
1.90	1.60	31.70	6.40	13.50	3.40	98.80
100.00	45.00	20.00	0.70	0.00	0.00	100.60
1.30	1.20	73.80	11.10	0.00	0.00	100.10
41.20	23.00	90.00	16.90	0.00	0.00	99.90
0.00	0.00	6.00	5.00	15.00	9.00	104.00
2.00	0.40	0.00	0.00	6.00	1.00	97.40
0.00	0.00	7.00	4.00	9.70	6.00	95.90
0.00	0.00	25.00	12.00	11.00	5.00	96.00
17.79	9.90	37.99	8.30	5.44	2.05	98.83

What is meant by the class "space" is the space that separates different types of blocks in each page of each magazine. For example, the space between lines and the space in graphs are not counted as space in our classification.

3.2. Device Description

The IBM PC class of computers has many resolutions that depend on the graphics board used. The most common boards are:

a) The Color Graphics Adapter (CGA).

b) The Enhanced Graphics Adapter (EGA).

The CGA has many modes of resolution. Some of these modes are for text only and some are for graphics and text. Since we need to display graphics, we chose the graphics modes. From these graphics modes, the mode with the highest number of displayed pels is mode 6 which can display 640 pels/line x 200 lines/screen x 2 colors/pel, where the two colors are black and white.

The EGA has the same modes of the CGA and more. The highest resolution it can display is 640 pels/line x 350 lines/screen x 16 colors/pel.

At the time this research started, the CGA was widely available while the EGA was at its second year and starting to be popular. This fact plus the fact that the investigation we did in section 3.1 showed that most of the library documents consisted of text and graphics, led us to choose the CGA at the start. The goal was to investigate applying the compression algorithms in the CGA with the text and graphics screens. Then, based on the result we get from this investigation, we will investigate the modification of the algorithms in the EGA. Due to time limitations, this research will not investigate the algorithms in the EGA; furthermore, in a library system we envision that the data will be sent in CGA mode 6 unless colors or photos are needed. This is due to the following reasons:

1. The CGA resolution is adequate and the size of the screen data is 1/7th of the size of the EGA screen.
2. If more than two colors are needed, the system can send these data in EGA mode after signaling the receiver of the change in resolution.
3. Although the EGA can display more text lines per page than the CGA, the quality of the text is good only if it displayed the same number of lines (25 text lines/page).

In the following part of the thesis, the resolution of the IBM PC is assumed to be CGA mode 6 unless otherwise specified. The compression and decompression times were measured on an IBM PC AT (6 MHz). Note that the maximum resolution of the new class of IBM machines (PS/2) is 640 x 480 x 256.

3.3. Procedures of the Research

The aim of this research is to experiment with the compression algorithms presented in the next chapters at the resolution described in the previous section. The following points will be examined in the research:

1. The compression factors calculated at this resolution using the different algorithms.

2. The class of images for which each algorithm works the best among the other algorithms.
3. The effect on the compression factor of dividing the screen into small blocks then compressing each block alone.
4. For the low resolution of the PC display, the effect on the compression factor in case of changing the method, its code, or both.

A very important point that should be kept in mind is the fact that, in the regular screen format, the background of the computer screen is black and the foreground is white. In regular papers, the reverse is true. Throughout this thesis, we will use the regular screen format unless otherwise specified.

3.4. Creation of the Image Data Base

The resolution of the IBM PC is a lot smaller than the CCITT low resolution (1728 x 1128). There are no standard images generated in this resolution available. To overcome this unavailability, we had to build our own image data base that represents the type of data we usually find in a library and that needs to be transmitted. The following guidelines were used in designing the data base:

a. We tried to match the screen size to the actual size of the data to be transmitted by letting each screen take what is equivalent to 25 lines in an A4 size paper. So, a paper with graphics that are equal in height to 50 lines will require two screens to represent it. Note that the text we generate will also differ from the text in a regular paper due to the fact that the spacing between lines is zero in CGA mode 6. In fact, in the graphics screen, each character takes 8x8 pels block and these blocks have no spacing between them. However, this does not mean that the characters will be connected to each other because in each character block the bottom or the upper line is empty.

b) For the horizontal resolution, we limited the part we took from the documents to the equivalent of 80 characters/line of text because this is the limit of the PC screen.

c) The CCITT standard documents do not represent very well the data we want to transmit. So, we created many other samples to be tested.

Appendix A contains a copy of this image data base.

3.5. Classification of the Image Data Base

To help us investigate the compression algorithms applicability in the screen and the best way to use them, images for the following classes of screens were generated:

1. Screens that imitate CCITT documents 1, 2, 4, 5, 6, and 8.
2. Screens that are full of graphics data.
3. Screens that are full of text.
4. Screens that are mixed of both text and graphics and sent as whole screens.
5. Screens that have one or more blocks of graphics.
6. Screens that can be considered as blocks of text and graphics and sent as blocks.
7. Screens that are not typical.
8. Screens to test power or limitations of the methods.

By having this extensive data base, we hope it will be a good test for the compression algorithms. From now on, each class will be assigned a group number according to its order above.

3.6. Results to be Analyzed

The images in the data bases were compressed then decompressed. The results of compressing each screen are:

a. Compression factor = original size/compressed size.

b. Compression time.

c. Decompression time.

The results of compressing the imitations of the CCITT documents were compared to published results of compressing these documents using CCITT standard techniques at facsimile resolution. To make the comparison more meaningful, the compression factor of compressing each document and not its parts was used in the comparison. This compression factor was normalized by dividing it by the compression factor of document no. 1.

3.7. Implementation Considerations

The following points are some general remarks about the code we wrote to simulate the algorithms:

1. The byte switching that the 8088 family uses makes accessing the screen buffer confusing if we want to access it as words. The reason of accessing words instead of bytes is to speed up the program execution.

2. An earlier version of the program for the one dimensional facsimile techniques translated the bits of the current line into a string where each pel is represented by a byte and the program was written to use this feature. Then the program was changed to its current form where the pels are accessed as bits in a word. Although the words and bits form is more complex, it gave about 40% decrease in compression time. This is due to the fact that the time spent in converting bits to string was a waste in the string version.

3. Writing the code in an optimized manner makes a big difference in both the size and speed of the final executable code. An optimization of the code resulted in 45% increase in speed of compression.

4. At early stages of the development, a big consideration was given to code optimization. Starting from the coding of the two dimensional technique, the big emphasis in optimizing was relaxed because it needed a lot of trials in order to find the most optimum form. This does not mean that the code was not optimized from that point on. It only means that we no longer try different formats of the code.

5. Most of the code was written in C language, but part of it was written in assembly language under the following conditions:

    a) This part of the code is executed a lot of times or it has a lot of looping. So, writing it in assembly language increases the speed of execution.

    b) The assembly language provides some commands that enhances the program, and no corresponding powerful commands are available in C language. Examples of these commands are the string instructions of the assembly language which provide a speed that cannot be reached in C because these string instructions are implemented by the hardware.

4. FACSIMILE CODING

4.1. Introduction

In this chapter, we will look at the use of the CCITT standard one- and two-dimensional facsimile compression techniques for compressing images in the data base described in section 3.4. The two standards were chosen because of the following reasons:

1. They are from the best (each in its dimension) techniques discussed in the literature.
2. By using them, we may provide the ability to connect the computer to facsimile machines.
3. A chip that has these two standards built in it was introduced. So, building a hardware system that uses these two standards is feasible.
4. To the best of our knowledge, no report of using these two coding techniques has been done for the same resolution we are working at.

The CCITT coding techniques have some features that are unnecessary to us, so we decided to drop these extra features. This resulted in our code not being exactly the CCITT code. In the following sections, we will describe the actual implementation of the codes and then give the corresponding results.

4.2. One Dimensional Compression Technique

For each line, this technique reads the runs of black and white, looks up the code of each run from the modified Huffman table, and then sends the code to the receiver or puts it in the compression buffer. This process is then repeated for each line till all lines are coded.

The steps of the compression algorithm are the following:

```
1. Initialize lines counter.

   Start on the first line.

2. Read first pel ($pel_0$) in the line.

   If ($pel_0$ is white)

       {insert the code of a black run of length zero in
        the compression buffer}.

   Set color to the color of $pel_0$.

   pels counter = 1.

3. While (the color does not change and end of line is not
   reached)

       {increment the pels counter}.

4. Put the code of the run of the current color in the compressed buffer.

5. If (the line ended)

       {if there are more lines}

             {"start on next line" GO TO 2}

        else

             {"the screen ended" GO TO 6}

   else

       {"the color changed within a line" GO TO 3}.

6. END.
```

The steps of decompression algorithms are the following:

```
1. Initialize lines counter.

   Start on the first line.
```

```
 2.  Initialize indexes of the compression and decompression
     buffers.

 3.  Read the compression buffer from left to right starting
     at its index and find the first bits to match a code for
     a black run.

 4.  Put the run corresponding to the matched code in the
     decompression buffer and adjust its index.

     Increment the index of the compression buffer by the length
     of the matched code.

 5.  If (decompressed data filled a line)

         GO TO ENDLINE.

 6.  Read the compression buffer from left to right starting
     at its index and find the first bits to match a code for
     a white run.

 7.  Put the run corresponding to the matched code in the de-
     compression buffer and adjust its index.

     Increment the index of the compression buffer by the length
     of the matched code.

 8.  If (decompressed data filled a line)

         GO TO ENDLINE.

 9.  GO TO 3.

10.  "ENDLINE":  Decrement lines counter.

                 If there are more lines GO TO 2.

11.  END.
```

For more details of the code, refer to Appendix B. This implementation of the code has the following differences with the CCITT standard for one-dimensional coding:

1. No minimum scan line time is assumed. Hence, no fill bits are used.

2. End of line code is not used. The compressor sends the size of the block at the beginning of the data, then the decompressor uses these data to step from line to line.
3. The screen has horizontal resolution of 640 pels. Hence, the run of 640 pels was used as a terminating word not as a make-up one. Without this, it will be necessary to send the code of a run equal to zero pels after the code for a run equal to 640 pels is sent.

The differences 1 and 2 above arose because the CCITT version of these points allows the compressor and the decompressor to synchronize and/or allows for mechanical limitations. These limitations are not present in the electronic library system. Hence, they will be disregarded. The end of line code is used in the two CCITT standards to correct the data if necessary. We assume that the communication software performs the error correction or that the communication channel is error free. Hence, no code for error correction is inserted.

The results of applying the one dimensional coding technique to the image data base are presented in Tables 4.1-4.8.

4.3. Two Dimensional Compression Technique

The CCITT two dimensional coding technique, titled MREAD, was used. The general concept of MREAD is that the changing elements in the coding line and the reference line take one out of three states. The code sent is optimized for these states. MREAD has the same concept we described in our review of [9]. For a complete description of MREAD, refer to [13]. In the following discussion, we will use terms and notations defined in [13].

Table 4.1. Results of compressing images in Group 1 using the CCITT one dimensional compression technique

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	C.F.[a] / T.C.F.	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
doc1a	0	0	639	199	8.03	9.27	0.87	214	99
doc1b	0	0	639	199	3.47	4.49	0.77	280	242
doc1c	0	0	639	199	15.15	20.62	0.73	192	50
doc2a	0	0	639	199	9.00	12.74	0.71	203	77
doc2b	0	0	639	199	8.12	10.91	0.74	209	83
doc2c	0	0	639	199	12.67	16.34	0.78	198	61
doc4a	0	0	639	199	1.96	2.56	0.77	368	428
doc4b	0	0	639	199	1.67	2.15	0.78	395	500
doc4c	0	0	639	87	1.74	2.24	0.78	170	214
doc51a	3	0	514	199	3.80	4.45	0.85	209	165
doc51b	0	0	511	199	5.52	6.61	0.84	187	110
doc51c	0	0	511	114	9.49	16.72	0.57	93	33
doc5ra	0	0	479	199	2.46	2.93	0.84	237	247
doc5rb	0	0	479	199	6.19	8.13	0.76	165	88
doc5rc	0	0	479	114	2.86	3.53	0.81	126	121
doc6a	0	0	639	199	4.77	7.01	0.68	231	149
doc6b	0	0	639	199	6.83	13.15	0.52	214	104
doc8	0	0	639	199	5.61	9.64	0.58	203	93
AVERAGE					6.07	8.53	0.74	216	159

---

[a] C.F. = Comprs. factor.
T.C.F. = Theort. comprs. factor.

Table 4.2. Results of compressing images in Group 2 using the CCITT one dimensional compression technique

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	$\frac{\text{C.F.}}{\text{T.C.F.}}$	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
frnch3a	0	0	639	199	5.23	7.34	0.71	220	127
flowchrt	0	0	639	199	4.18	5.39	0.78	247	176
electrc	0	0	639	199	2.05	3.44	0.60	318	352
ordrfrm	0	0	639	199	4.13	5.37	0.77	258	192
frnch1a	0	0	639	199	5.90	7.92	0.74	231	132
doc2a	0	0	639	199	9.00	20.74	0.43	204	77
doc2b	0	0	639	199	8.12	10.91	0.74	208	88
AVERAGE					5.52	8.73	0.68	241	163

Table 4.3. Results of compressing images in Group 3 using the CCITT one dimensional compression technique

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	$\frac{\text{C.F.}}{\text{T.C.F.}}$	Comprs. time (1/100th s)	Dcomprs. time (1/100th s)
romtxt	0	0	639	199	1.46	1.91	0.76	434	582
frnch2a	0	0	639	199	2.07	2.66	0.78	351	401
page1	0	0	639	199	3.19	4.04	0.79	291	258
doc1-2	0	0	639	199	3.38	4.43	0.76	280	248
cprog	0	0	639	199	5.56	7.22	0.77	236	149
doc1b	0	0	639	199	3.47	4.49	0.77	280	242
doc4a	0	0	639	199	1.96	2.56	0.77	362	428
doc4b	0	0	639	199	1.67	2.15	0.78	396	500
AVERAGE					2.84	3.68	0.77	329	351

Table 4.4. Results of compressing images in Group 4 using the CCITT one dimensional compression technique

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	C.F. / T.C.F.	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
pdraw3	0	0	639	199	4.09	4.84	0.85	253	192
science1	0	0	639	199	3.58	4.12	0.87	263	214
science2	0	0	639	199	2.69	3.21	0.84	308	297
doc51a	0	0	514	199	3.80	4.45	0.85	209	165
AVERAGE					3.54	4.16	0.85	258	217

Table 4.5. Results of compressing images in Group 5 using the CCITT one dimensional compression technique

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	C.F. / T.C.F.	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
opamp1	160	0	639	158	5.80	7.31	0.79	231	71
opamp2	0	0	639	190	5.35	6.47	0.83	225	137
ecl1	72	7	551	166	8.88	12.51	0.71	121	44
ecl2	0	7	607	190	8.09	10.84	0.75	181	77
netwrk	16	9	623	187	5.10	7.58	0.67	192	121
table1	0	13	639	147	3.17	3.80	0.83	181	159
usa1	56	24	519	164	10.43	17.24	0.60	99	33
doc51a	36	48	483	115	5.79	9.06	0.64	55	27
doc5rb	28	43	475	169	5.31	7.19	0.74	105	61
lotssin	88	22	631	165	3.00	5.08	0.59	171	160
frnch3b	0	0	639	71	5.40	7.96	0.68	77	44
barchrt	30	10	333	145	4.50	6.77	0.66	71	44
barchrt	30	10	237	60	2.45	5.48	0.45	22	22
barchrt	32	68	335	145	5.57	9.70	0.57	38	27
test2	120	15	455	120	5.08	6.79	0.75	60	38
test3	120	15	455	120	4.54	5.33	0.85	66	43
test4	120	15	455	120	4.05	4.72	0.86	66	49
test5	120	15	487	120	3.95	4.41	0.90	77	60
diag1	70	26	453	120	5.83	7.92	0.74	66	33
diag2	42	42	393	108	7.03	9.20	0.76	38	17
diag3	210	18	449	131	1.07	4.02	0.27	99	138
diag4	108	14	443	88	5.03	6.13	0.82	44	28
diag5	68	5	467	102	7.44	16.89	0.44	60	28
diag5s	208	28	479	98	5.46	11.85	0.46	33	17
diag6	40	9	279	76	5.03	10.84	0.46	33	22
diag6	22	109	405	141	6.35	15.21	0.42	22	11
diag6	22	9	405	141	8.29	19.13	0.43	82	33
netwrk2	136	62	391	136	2.88	5.30	0.54	38	33
pdraw1	0	70	287	150	4.09	4.91	0.83	44	33
usa2	202	26	329	61	3.27	3.94	0.83	11	11
usa2	164	92	403	162	5.38	7.46	0.72	33	16
doc51b	24	19	471	51	7.62	17.35	0.44	22	11
science3	0	80	127	196	2.94	3.46	0.85	32	33
science3	456	12	535	66	2.29	2.90	0.79	11	11
AVERAGE					5.19	8.38	0.67	80	50

Table 4.6. Results of compressing images in Group 6 using the CCITT one dimensional compression technique

Image	x1	y1	x2	y2	Comprs. factor	Cmprs. time (1/100th s)	Dcmprs. time (1/100th s)
pdraw1.pic	0	0	559	150	3.96	170	132
pdraw2.pic	0	0	575	152	3.36	186	159
pdraw3.pic	0	0	575	191	3.56	230	192
pdraw3.pic	16	0	559	39	1.48	72	99
pdraw3.pic	0	70	287	150	4.09	50	33
pdraw3.pic	380	77	571	152	2.66	33	33
pdraw3.pic	48	160	575	191	3.65	33	33
pdraw3.pic	0	0	639	199	4.09	258	192
Compression factor using 4 blocks					3.36		

Table 4.7. Results of compressing images in Group 7 using the CCITT one dimensional compression technique

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	C.F. / T.C.F.	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
bignames	0	0	639	199	1.38	2.10	0.66	429	566
sun	0	0	639	199	2.62	3.68	0.71	297	291
hazard	0	0	639	199	2.38	3.44	0.69	307	324
mansc1	0	0	639	199	1.96	2.62	0.75	340	390
mansc2	0	0	639	199	2.74	3.48	0.79	285	275
fig2	0	0	639	199	1.41	6.31	0.22	346	439
fig4	0	0	639	199	2.86	6.76	0.42	275	247
fig6	0	0	639	199	3.43	4.85	0.71	263	214
fig7	0	0	639	199	5.04	7.71	0.65	231	143
fig8	0	0	639	199	3.10	4.50	0.69	275	242
AVERAGE					2.69	4.55	0.63	305	313

Table 4.8. Results of compressing images in Group 8 using the CCITT one dimensional compression technique

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	$\frac{\text{C.F.}}{\text{T.C.F.}}$	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
blok3	0	0	639	199	27.22	134.12	0.20	176	22
blok6	0	0	639	199	4.63	16.11	0.29	225	143
boxes	0	0	639	199	12.12	51.39	0.24	192	61
lines	0	0	639	199	7.27	48.38	0.15	214	104
test1	120	15	455	120	10.91	56.29	0.19	54	17
usamap	72	28	551	164	(Comprs. factor < 1, not applicable)				
AVERAGE					12.43	61.26	0.21	172	69

Reference [13] gave details and a flowchart of the compression and we provided details of the process of decompression in the flowchart in Figure 4.1.

4.4. MREAD Implementation and Results

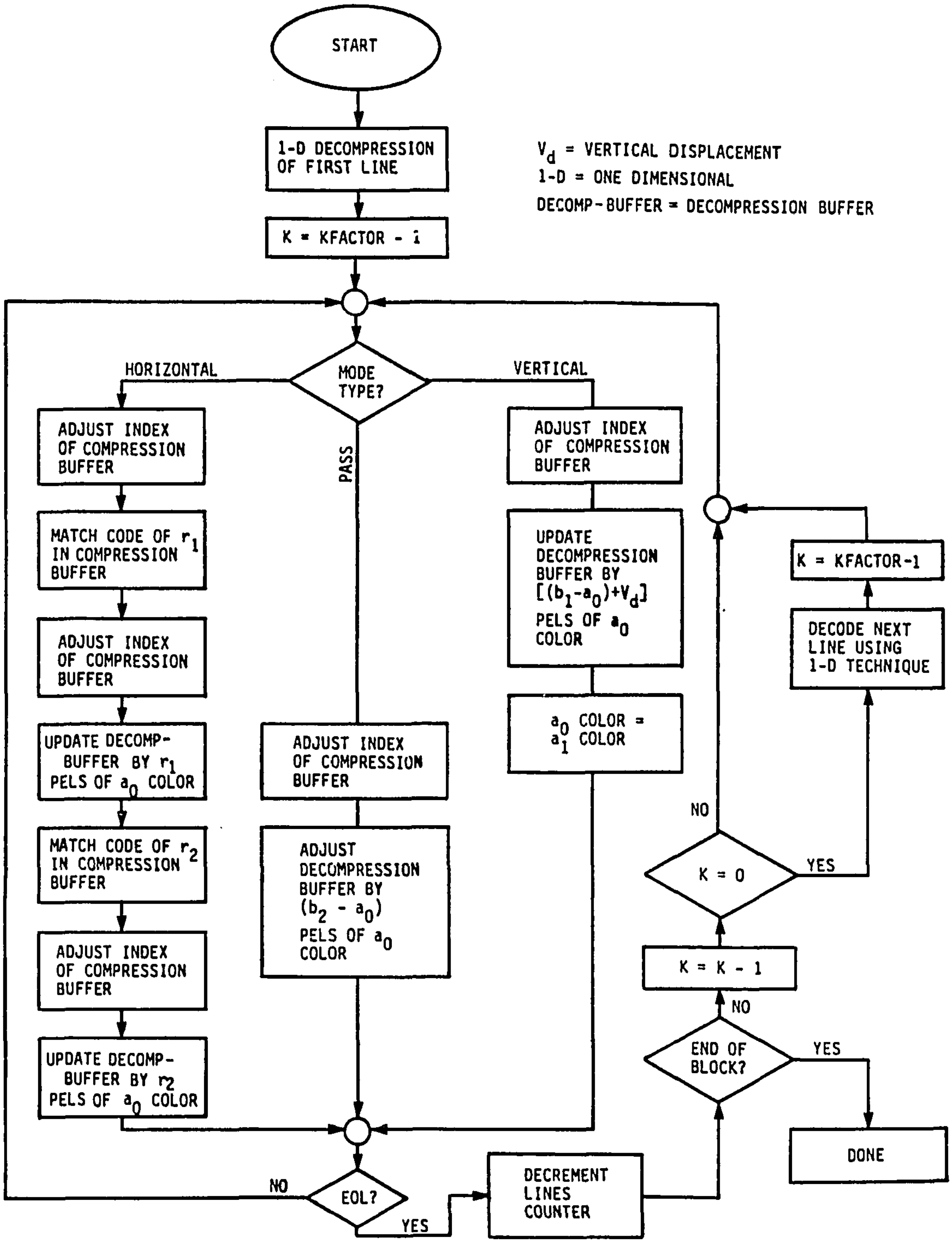
The code for MREAD is presented in Appendix C. A close look at the code combined with our experience while debugging the program suggests that the code matching part might be improved in speed if we write the matching in a tree-like form, i.e., using IF() THEN {} ELSE {} and nesting these conditions. Such a code was tried and gave an average of 9% decrease in decompress time.

MREAD suggested using $k = 2$ to help in recovering from errors which decrease the compression factor. If no error recovery is needed, $k = \infty$ can be used. This will give higher compression factor. To get $k = \infty$, it is only necessary to let KFACTOR be 201 in the programs listed in Appendix C.

MREAD was modified by the modification described for the one dimensional coding technique in section 2. Note that although MREAD has minimum scan line time specification, it has no fill bits.

The results of compressing the data base images for the case of $k = 2$ and $k = \infty$ are given in Tables 4.9-4.16 and Tables 4.17-4.24, respectively. The times are obtained by using a tree-like code.

Figure 4.1. Flow diagram of the decompression process using the CCITT two dimensional compression technique



V_d = VERTICAL DISPLACEMENT

1-D = ONE DIMENSIONAL

DECOMP-BUFFER = DECOMPRESSION BUFFER

Table 4.9. Results of compressing images in Group 1 using the CCITT two dimensional compression technique with k = 2

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
doc1a	0	0	639	199	8.71	286	165
doc1b	0	0	639	199	3.41	400	325
doc1c	0	0	639	199	17.21	248	104
doc2a	0	0	639	199	11.04	258	126
doc2b	0	0	639	199	9.98	264	138
doc2c	0	0	639	199	15.27	248	110
doc4a	0	0	639	199	1.89	544	522
doc4b	0	0	639	199	1.60	598	604
doc4c	0	0	639	87	1.67	259	258
doc51a	3	0	514	199	3.96	291	220
doc51b	0	0	511	199	6.33	247	154
doc51c	0	0	511	114	12.70	120	55
doc5ra	0	0	479	199	2.45	346	307
doc5rb	0	0	479	199	7.67	220	127
doc5rc	0	0	479	114	2.87	181	154
doc6a	0	0	639	199	6.39	308	181
doc6b	0	0	639	199	9.62	274	143
doc8	0	0	639	199	9.14	259	132
AVERAGE					7.33	297	213

Table 4.10. Results of compressing images in Group 2 using the CCITT two dimensional compression technique with k = 2

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
frnch3a	0	0	639	199	7.29	297	176
flowchrt	0	0	639	199	4.75	335	231
electrc	0	0	639	199	2.04	478	445
ordrfrm	0	0	639	199	4.29	362	269
frnch1a	0	0	639	199	6.36	313	198
doc2a	0	0	639	199	11.04	258	126
doc2b	0	0	639	199	9.98	264	137
AVERAGE					6.54	330	226

Table 4.11. Results of compressing images in Group 3 using the CCITT two dimensional compression technique with k = 2

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
rcmtxt	0	0	639	199	1.38	670	698
frnch2a	0	0	639	199	2.01	533	505
page1	0	0	639	199	3.11	407	335
doc1-2	0	0	639	199	3.25	400	324
cprog	0	0	639	199	5.42	324	220
doc1b	0	0	639	199	3.41	401	324
doc4a	0	0	639	199	1.89	544	527
doc4b	0	0	639	199	1.60	599	604
AVERAGE					2.76	485	442

Table 4.12. Results of compressing images in Group 4 using the CCITT two dimensional compression technique with k = 2

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
pdraw3	0	0	639	199	4.32	352	258
science1	0	0	639	199	3.87	379	280
science2	0	0	639	199	2.63	445	384
doc51a	3	0	514	199	3.96	292	214
AVERAGE					3.70	367	284

Table 4.13. Results of compressing images in Group 5 using the CCITT two dimensional compression technique with k = 2

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
opamp1	160	0	639	158	7.12	170	99
opamp2	0	0	639	190	5.95	297	187
ec11	72	7	551	166	11.45	153	71
ec12	0	7	607	190	9.87	236	121
netwrk	16	9	623	187	6.79	264	159
table1	0	13	639	147	3.95	258	198
usa1	56	24	519	164	11.90	126	55
doc51a	36	48	483	115	7.06	65	38
doc5rb	28	43	475	169	6.94	137	82
lotssin	88	22	631	165	3.55	241	187
usamap	72	28	551	164	0.63	747	851
frnch3b	0	0	639	71	7.60	104	60
barchrt	30	10	333	145	6.47	99	60
barchrt	30	10	237	60	4.11	33	22
barchrt	32	68	335	145	6.85	55	33
test1	120	15	455	120	18.01	72	27
test2	120	15	455	120	5.79	83	49
test3	120	15	455	120	5.02	88	55
test4	120	15	455	120	4.41	94	60
test5	120	15	487	120	4.24	104	72
diag1	70	26	453	120	7.16	83	50
diag2	42	42	393	108	7.79	49	22
diag3	210	18	449	131	1.26	153	154
diag4	108	14	443	88	5.63	60	38
diag5	68	5	467	102	10.80	83	39
diag5s	208	28	479	98	7.69	44	28
diag6	40	9	279	76	7.45	38	22
diag6	22	109	405	141	9.38	28	16
diag6	22	9	405	141	12.69	105	50
netwrk2	136	62	391	136	3.81	55	44
pdraw1	0	70	287	150	4.58	60	44
usa2	202	26	329	61	3.68	11	5
usa2	164	92	403	162	6.07	39	22
doc51b	24	19	471	51	11.20	27	11
science3	0	80	127	196	3.20	49	38
science3	456	12	535	66	2.33	16	17
AVERAGE					6.58	120	86

Table 4.14. Results of compressing images in Group 6 using the CCITT two dimensional compression technique with k = 2

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
pdraw1	0	0	559	150	4.01	236	176
pdraw2	0	0	575	152	3.55	258	203
pdraw3	0	0	575	191	3.76	324	247
pdraw3	16	0	559	39	1.41	110	116
pdraw3	0	70	287	150	4.58	61	39
pdraw3	380	77	571	152	3.56	44	32
pdraw3	48	160	575	191	3.70	50	39
pdraw3	0	0	639	199	4.32	352	252
Compression factor using 4 blocks							

Table 4.15. Results of compressing images in Group 7 using the CCITT two dimensional compression technique with k = 2

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
bignames	0	0	639	199	1.48	659	670
sun	0	0	639	199	2.89	423	352
hazard	0	0	639	199	2.46	439	379
mansc1	0	0	539	199	2.10	517	472
mansc2	0	0	639	199	3.10	417	340
fig2	0	0	639	199	1.77	538	495
fig4	0	0	639	199	4.02	379	280
fig6	0	0	639	199	3.87	357	263
fig7	0	0	639	199	5.57	302	186
fig8	0	0	639	199	3.47	379	291
AVERAGE					3.07	441	373

Table 4.16. Results of compressing images in Group 8 using the CCITT two dimensional compression technique with k = 2

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
blok3	0	0	639	199	43.17	225	77
blok6	0	0	639	199	7.78	302	176
boxes	0	0	639	199	20.55	248	104
lines	0	0	639	199	11.76	280	148
AVERAGE					20.82	264	126

Table 4.17. Results of compressing images in Group 1 using the CCITT two dimensional compression technique with k = ∞

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	$\frac{\text{C.F.}}{\text{T.C.F.}}$	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
doc1a	0	0	639	199	9.57	10.36	0.92	346	247
doc1b	0	0	639	199	3.40	3.88	0.88	511	439
doc1c	0	0	639	199	19.54	26.55	0.74	286	176
doc2a	0	0	639	199	14.30	18.96	0.75	302	193
doc2b	0	0	639	199	12.75	17.57	0.73	308	203
doc2c	0	0	639	199	18.83	27.37	0.69	280	176
doc4a	0	0	639	199	1.87	2.09	0.89	730	693
doc4b	0	0	639	199	1.56	1.69	0.92	808	786
doc4c	0	0	639	87	1.64	1.80	0.91	346	335
doc5la	3	0	514	199	4.23	5.28	0.80	368	297
doc5lb	0	0	511	199	7.60	10.18	0.75	297	220
doc5lc	0	0	511	114	18.81	35.12	0.54	137	88
doc5ra	0	0	479	199	2.50	2.79	0.90	450	401
doc5rb	0	0	479	199	10.10	13.79	0.73	258	186
doc5rc	0	0	479	114	2.90	3.36	0.86	236	209
doc6a	0	0	639	199	9.57	13.06	0.73	368	264
doc6b	0	0	639	199	16.10	25.71	0.63	318	214
doc8	0	0	639	199	22.99	32.69	0.70	308	197
AVERAGE					9.90	14.01	0.78	370	296

Table 4.18. Results of compressing images in Group 2 using the CCITT two dimensional compression technique with k = ∞

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	$\frac{\text{C.F.}}{\text{T.C.F.}}$	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
frnch3a	0	0	639	199	12.05	14.70	0.82	357	253
flowchrt	0	0	639	199	5.56	6.39	0.87	412	330
electrc	0	0	639	199	2.04	3.66	0.56	631	599
ordrfrm	0	0	639	199	4.77	5.74	0.83	467	385
frnch1a	0	0	639	199	7.16	9.41	0.76	385	286
doc2a	0	0	639	199	14.30	19.81	0.72	296	192
doc2b	0	0	639	199	12.75	18.33	0.70	313	209
AVERAGE					8.38	11.00	0.75	409	322

Table 4.19. Results of compressing images in Group 3 using the CCITT two dimensional compression technique with k = ∞

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	$\frac{\text{C.F.}}{\text{T.C.F.}}$	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
romtxt	0	0	639	199	1.40	1.57	0.89	900	890
frnch2a	0	0	639	199	1.96	2.11	0.93	698	660
page1	0	0	639	199	3.25	3.85	0.84	528	456
doc1-2	0	0	639	199	3.36	4.13	0.81	517	445
cprog	0	0	639	199	5.68	7.49	0.76	406	319
doc1b	0	0	639	199	3.40	4.26	0.80	511	440
doc4a	0	0	639	199	1.87	2.14	0.87	725	692
doc4b	0	0	639	199	1.56	1.70	0.92	807	785
AVERAGE					2.81	3.41	0.85	637	586

Table 4.20. Results of compressing images in Group 4 using the CCITT two dimensional compression technique with k = ∞

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	C.F. / T.C.F.	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
pdraw3	0	0	639	199	4.84	5.31	0.91	439	352
science1	0	0	639	199	4.19	4.85	0.86	467	390
science2	0	0	639	199	2.61	2.99	0.87	582	522
doc51a	0	0	514	199	4.23	5.10	0.83	363	297
AVERAGE					3.97	4.56	0.87	463	390

Table 4.21. Results of compressing images in Group 5 using the CCITT two dimensional compression technique with k = ∞

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	C.F. / T.C.F.	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
opamp1	160	0	639	158	9.07	10.08	0.90	204	143
opamp2	0	0	639	190	6.85	7.66	0.89	357	269
ecl1	72	7	551	166	15.82	22.10	0.72	181	115
ecl2	0	7	607	190	12.51	15.99	0.78	275	182
netwrk	16	9	623	187	9.87	13.51	0.73	313	226
table1	0	13	639	147	5.21	6.31	0.83	324	264
usa1	56	24	519	164	13.40	24.20	0.55	148	94
doc5rb	36	48	483	115	9.16	12.21	0.75	88	60
doc5rb	28	43	475	169	9.81	13.40	0.73	165	115
lotssin	88	22	631	165	4.38	7.17	0.61	302	247
frnch3b	0	0	639	71	12.42	20.48	0.61	126	88
barchrt	30	10	333	145	11.38	18.10	0.63	116	83
barchrt	30	10	237	60	12.01	68.63	0.17	38	28
barchrt	32	68	335	145	8.88	13.69	0.65	66	49
test2	120	15	455	120	6.75	8.21	0.82	94	71
test3	120	15	455	120	5.59	7.10	0.79	104	77
test4	120	15	455	120	4.86	6.18	0.79	110	88
test5	120	15	487	120	4.56	5.67	0.80	126	99
diag1	70	26	453	120	8.99	13.08	0.69	99	71
diag2	42	42	393	108	9.07	15.00	0.60	60	38
diag3	210	18	449	131	1.52	4.27	0.36	204	192
diag4	108	14	443	88	6.32	8.22	0.77	71	50
diag5	68	5	467	102	18.46	28.90	0.64	99	66
diag5s	208	28	479	98	12.76	18.15	0.70	55	38
diag6	40	9	279	76	13.88	21.61	0.64	44	33
diag6	22	109	405	141	17.70	41.35	0.43	33	16
diag6	22	9	405	141	26.19	45.54	0.58	120	77
netwrk2	136	62	391	136	5.78	8.43	0.69	66	49
pdraw1	0	70	287	150	5.34	6.16	0.87	71	55
usa2	202	26	329	61	3.94	4.80	0.82	11	11
usa2	164	92	403	162	7.35	9.78	0.75	50	33
doc51b	24	19	471	51	20.82	52.41	0.40	38	22
science3	0	80	127	196	3.40	4.36	0.78	60	50
science3	456	12	535	66	2.43	3.02	0.80	22	16
AVERAGE					9.60	16.64	0.68	125	92

Table 4.22. Results of compressing images in Group 6 using the CCITT two dimensional compression technique with k = ∞

Image	x1	y1	x2	y2	Comprs. factor	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
pdraw1	0	0	559	150	4.34	297	242
pdraw2	0	0	575	152	3.99	330	275
pdraw3	0	0	575	191	4.19	401	335
pdraw3	16	0	559	39	1.45	148	148
pdraw3	0	70	287	150	5.34	72	55
pdraw3	380	77	571	152	5.47	60	44
pdraw3	48	160	575	191	3.94	66	504
pdraw3	0	0	639	199	4.84	439	352

Compression factor using 4 blocks 4.18

Table 4.23. Results of compressing images in Group 7 using the CCITT two dimensional compression technique with k = ∞

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	C.F. / T.C.F.	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
bignames	0	0	639	199	1.59	1.81	0.88	873	851
sun	0	0	639	199	3.20	4.46	0.72	527	456
hazard	0	0	639	199	2.53	3.43	0.74	560	494
mansc1	0	0	639	199	2.28	2.79	0.82	665	610
mansc2	0	0	639	199	3.61	4.69	0.77	533	456
fig2	0	0	639	199	2.40	4.35	0.55	686	637
fig4	0	0	639	199	6.78	12.23	0.55	466	368
fig6	0	0	639	199	4.34	6.76	0.64	445	357
fig7	0	0	639	199	6.29	10.75	0.59	363	269
fig8	0	0	639	199	3.94	6.30	0.63	466	384
AVERAGE					3.70	5.76	0.69	558	488

Table 4.24. Results of compressing images in Group 8 using the CCITT two dimensional compression technique with k = ∞

Image	x1	y1	x2	y2	Comprs. factor	Theort. comprs. factor	$\frac{C.F.}{T.C.F.}$	Comprs. time (1/100th s)	Dcmprs. time (1/100th s)
blok3	0	0	639	199	109.03	359.73	0.30	252	138
blok6	0	0	639	199	24.54	142.33	0.17	363	247
boxes	0	0	639	199	69.57	813.68	0.09	280	171
lines	0	0	639	199	32.07	191.23	0.17	324	214
test1	120	15	455	120	49.60	99.19	0.50	77	44
usamap	72	28	551	164	1.56	7.13	0.22	1011	962
AVERAGE					47.73	268.88	0.24	385	296

4.5. Entropy Calculation of the One Dimensional Model

The one dimensional coding can be represented as a first order Markov chain as in Figure 4.2. The per pel entropy h_{WB} is given in [6] as follows:

$$h_{WB} = P_W \frac{H_W}{r_W} + P_B \frac{H_B}{r_B} \tag{4.1}$$

where:

P_W = probability of white pels

P_B = probability of black pels

H_W = white run-length entropy

$$= - \sum_{i=0}^{N} p_{wi} \cdot \log_2 p_{wi} \tag{4.2}$$

H_B = black run-length entropy

$$= - \sum_{i=0}^{N} p_{bi} \cdot \log_2 p_{bi} \tag{4.3}$$

p_{wi} = probability of run-length of i white pels

p_{bi} = probability of run-lengths of i black pels

r_W = average white run-length in pels = $\Sigma\, i \cdot p_{wi}$ (4.4)

r_B = average black run-length in pels = $\Sigma\, i \cdot p_{bi}$ (4.5)

Note that:

$$P_W + P_B = 1 \tag{4.6}$$

$$\Sigma\, p_{wi} = 1 \tag{4.7}$$

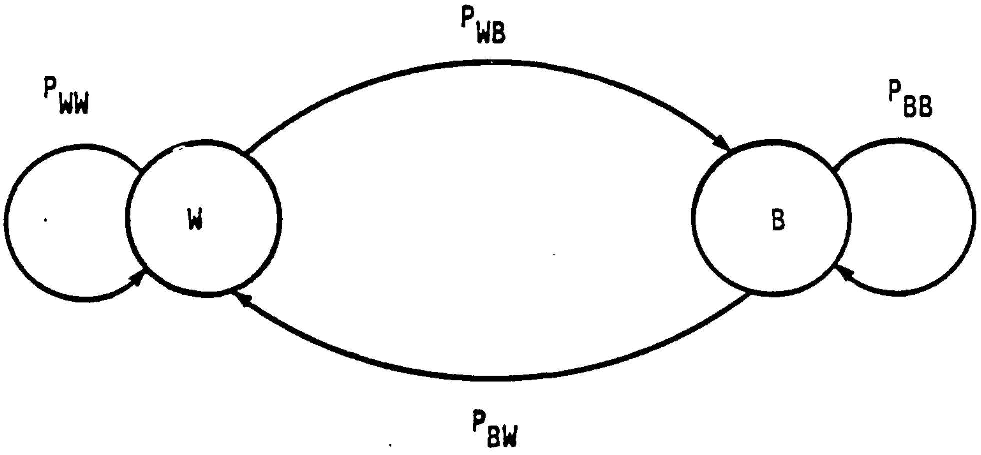


Figure 4.2. A first order Markov model for the CCITT one dimensional coding technique

$$\Sigma\, p_{bi} = 1 \tag{4.8}$$

To get p_W and p_B, we solve the matrix question:

$$[p_W\ p_B] \begin{bmatrix} p_{ww} & p_{wb} \\ p_{bw} & p_{bb} \end{bmatrix} = \begin{bmatrix} p_W \\ p_B \end{bmatrix} \tag{4.9}$$

Then, we get the following equations:

$$p_W = \frac{p_{wb}}{p_{wb} + p_{bw}} \tag{4.10}$$

$$p_B = \frac{p_{bw}}{p_{wb} + p_{bw}} \tag{4.11}$$

Substituting from (4.10) and (4.11) in (4.1), we get

$$h_{WB} = \frac{H_W + H_B}{P_W + P_B} \tag{4.12}$$

The maximum theoretical compression factor Q_{max} is defined as

$$Q_{max} = \frac{1}{h_{WB}} = \frac{r_W + r_B}{H_W + H_B} \tag{4.13}$$

Reference [12] applied CCITT one dimensional coding technique to the 8 CCITT documents and gave the result of r_W, r_B, H_W, H_B, Q_{max}, and actual compression factor in Table IV of the reference.

The result of calculating the Q_{max} of the data base is included in Tables 4.1-4.8.

Figures 4.3-4.11 show the distribution of the frequency of the run-lengths for a sample of images from the data base. Runs greater than 63 were broken into two runs as described by the standard.

4.6. Entropy Calculation of the Two Dimensional Model

Reference [13] did not calculate the entropy for the 8 CCITT documents. Reference [9], which has the same principles of using three states, did. Besides, the compression factors in [9] are comparable to those of MREAD. So, we will calculate the entropy and Q_{max} using a modified version of the model given in [9]. The model we will use is valid only for the case of $k = \infty$.

We assume that each of the three states is independent of the other states. Hence, the entropy per pel H_{pel} is given by

$$H_{pel} = \frac{H_s}{B_s} = \frac{1}{B_s} \sum_{j=1}^{3} P(S_j)H_j \tag{4.14}$$

where

H_s = average entropy per state

B_s = average number of pels per state

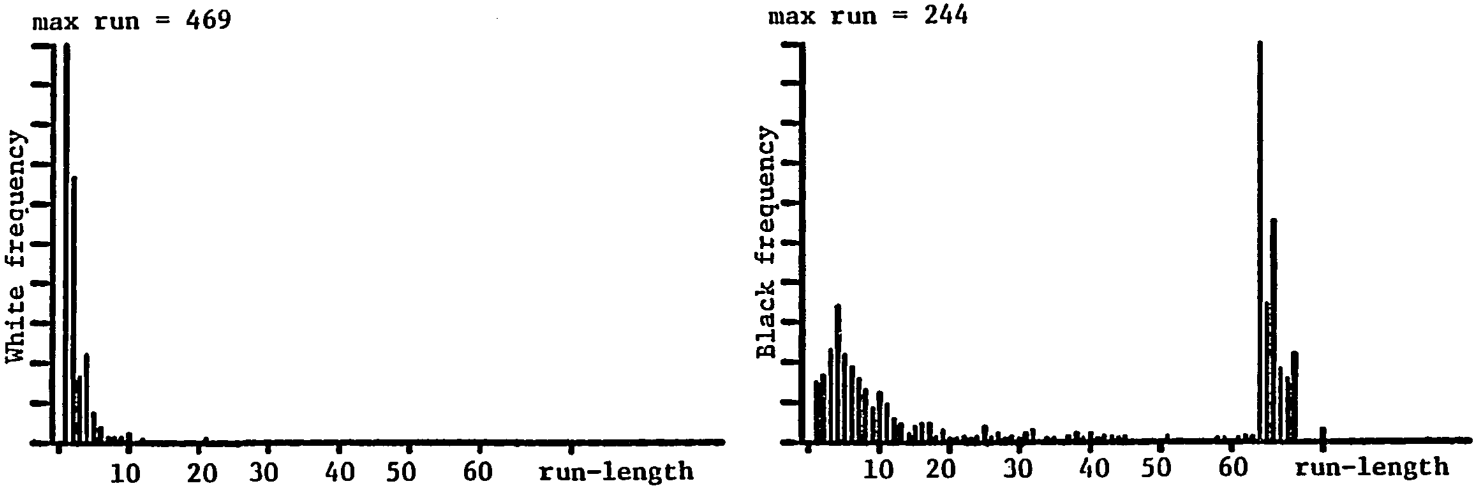
H_j = entropy of state S_j.

The entropies of the three states are given by

$$H_1 = -\log P(S_1) + H_d \tag{4.15}$$

$$H_2 = -\log P(S_2) \tag{4.16}$$

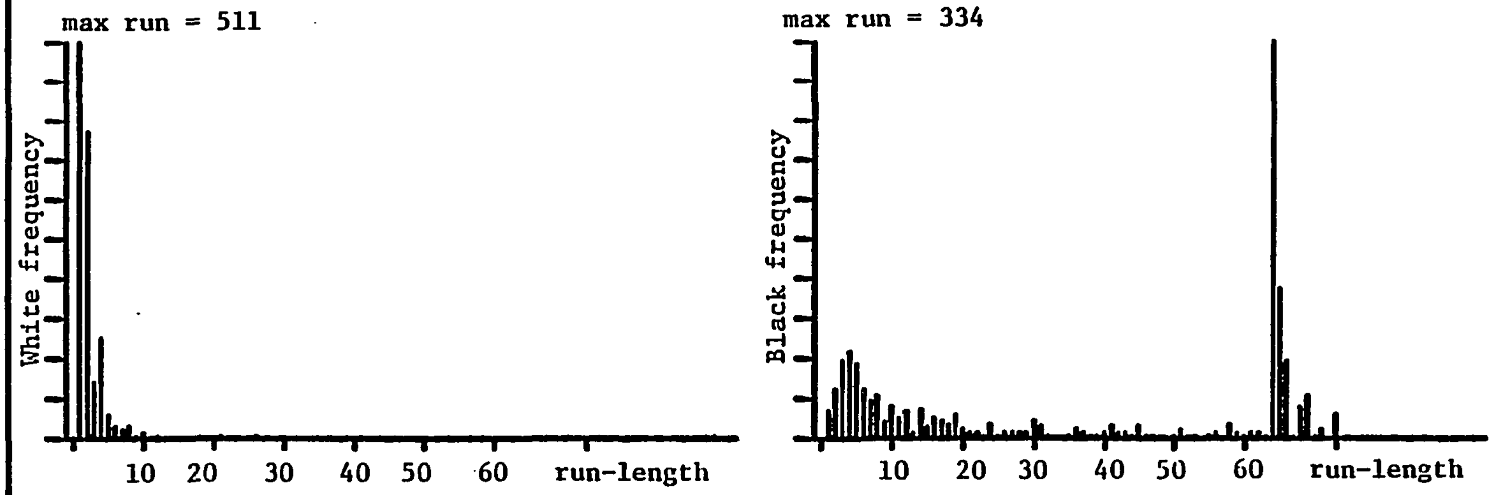
$$H_3 = -\log P(S_3) + H_{11} + H_{12} \tag{4.17}$$



(a) White run-length distribution

(b) Black run-length distribution

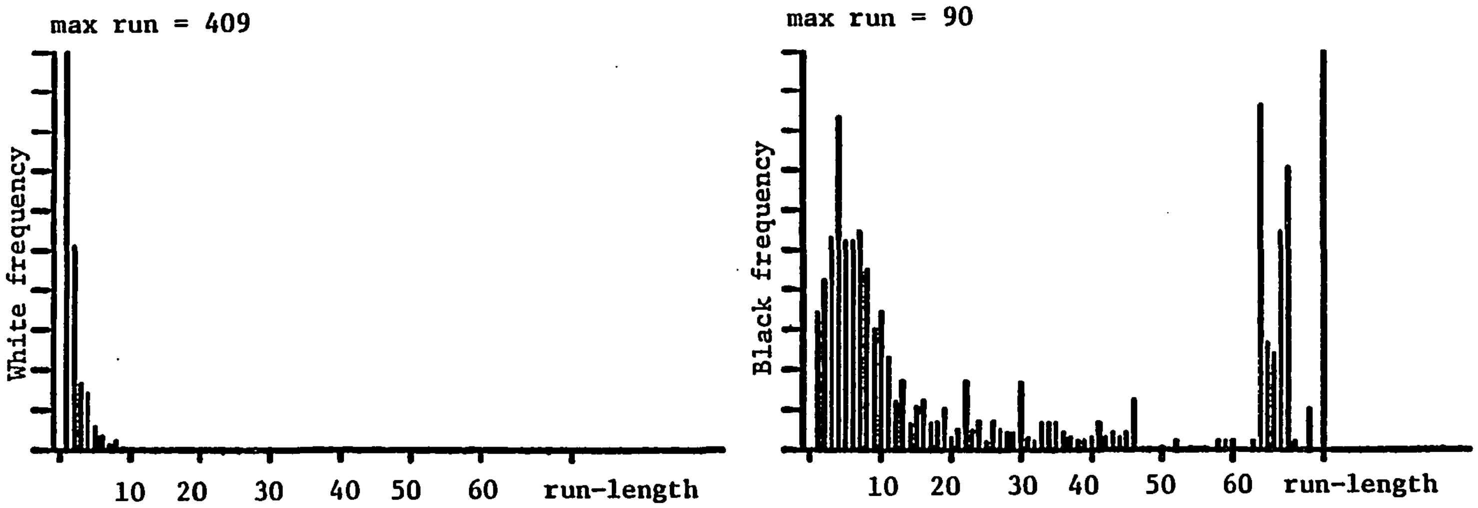
Figure 4.3. Frequency distribution of image "doc2a"



(a) White run-length distribution

(b) Black run-length distribution

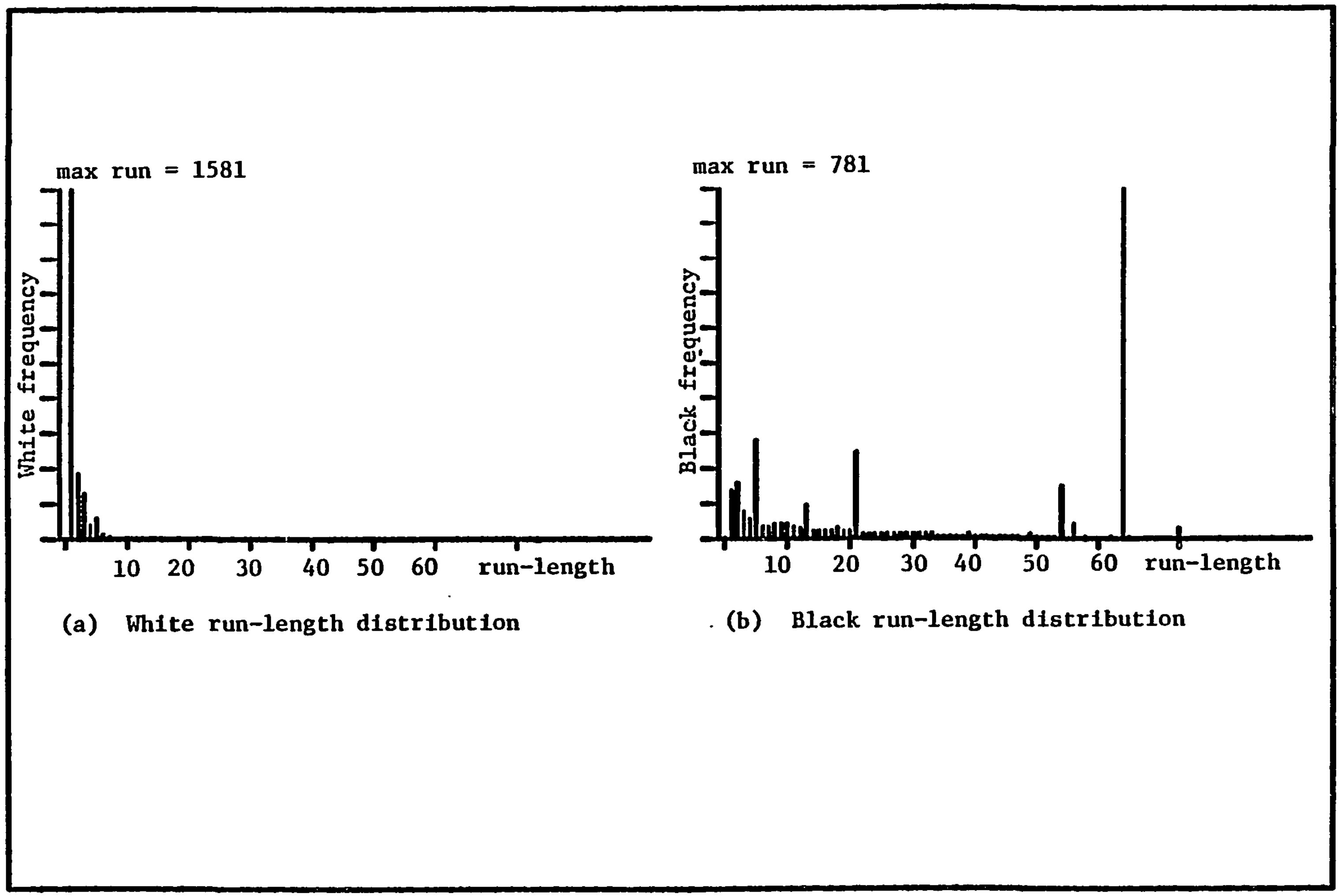
Figure 4.4. Frequency distribution of image "doc2b"



(a) White run-length distribution

(b) Black run-length distribution

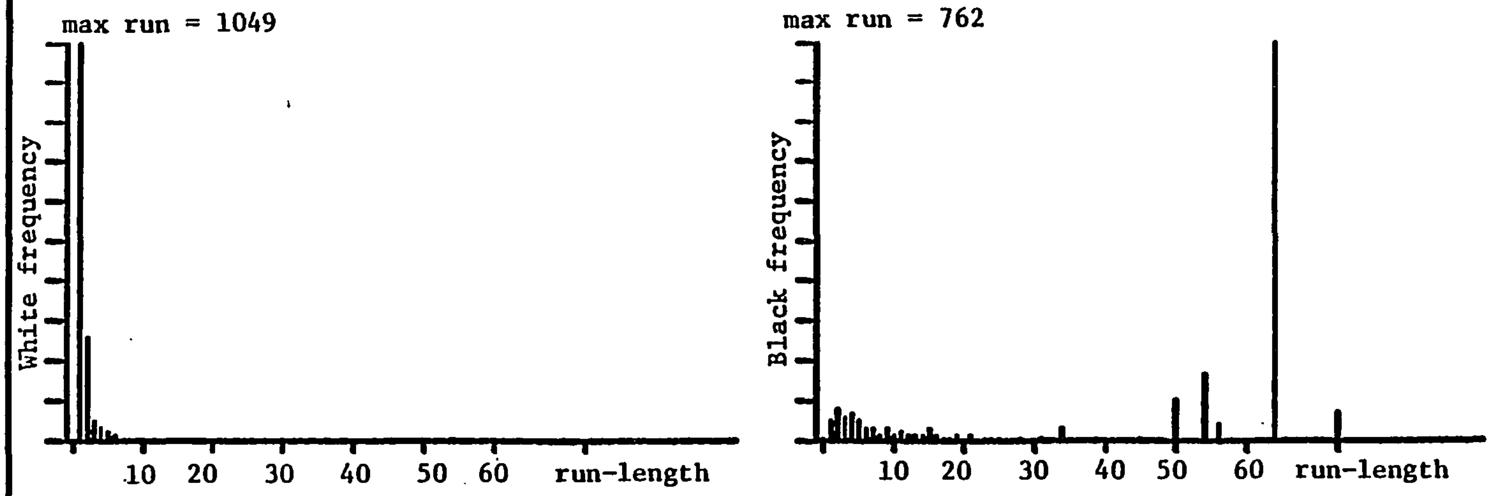
Figure 4.5. Frequency distribution of image "doc2c"



(a) White run-length distribution

(b) Black run-length distribution

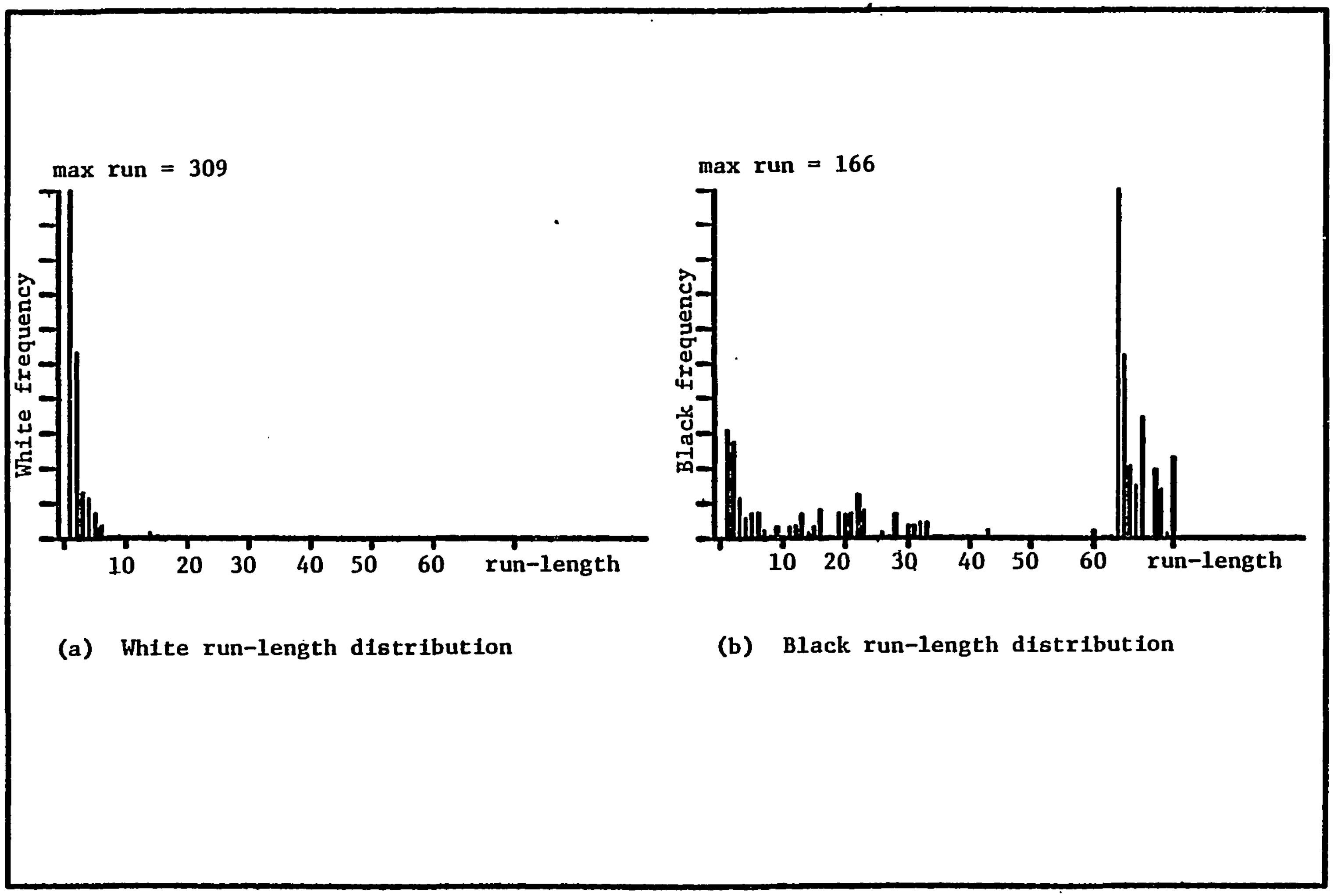
Figure 4.6. Frequency distribution of image "doc6a"



(a) White run-length distribution

(b) Black run-length distribution

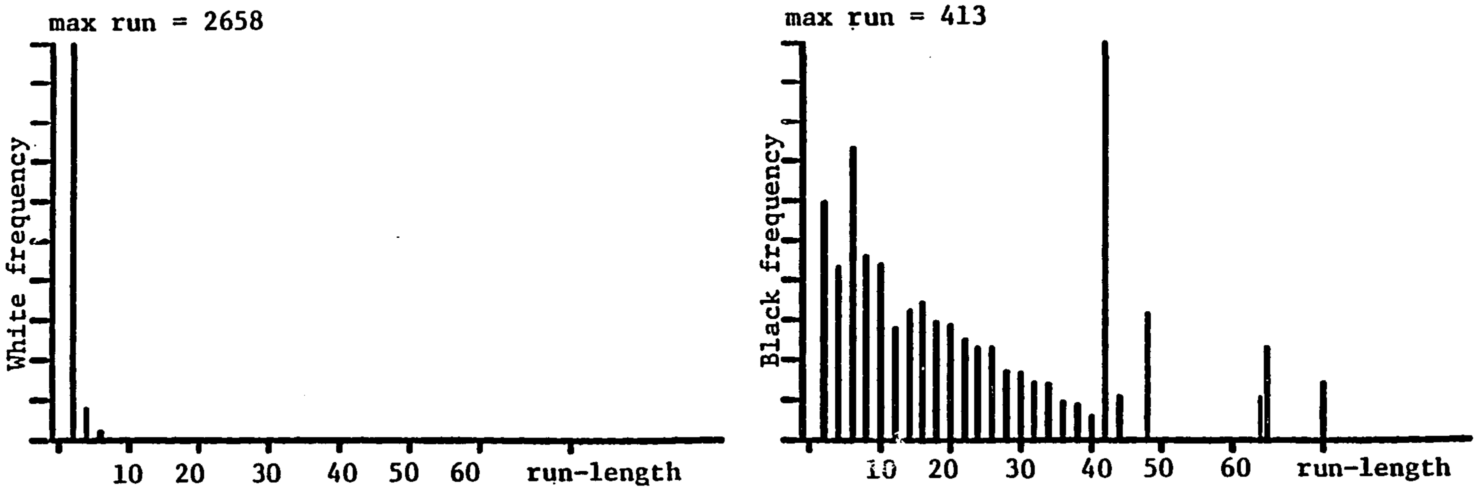
Figure 4.7. Frequency distribution of image "doc6b"



(a) White run-length distribution

(b) Black run-length distribution

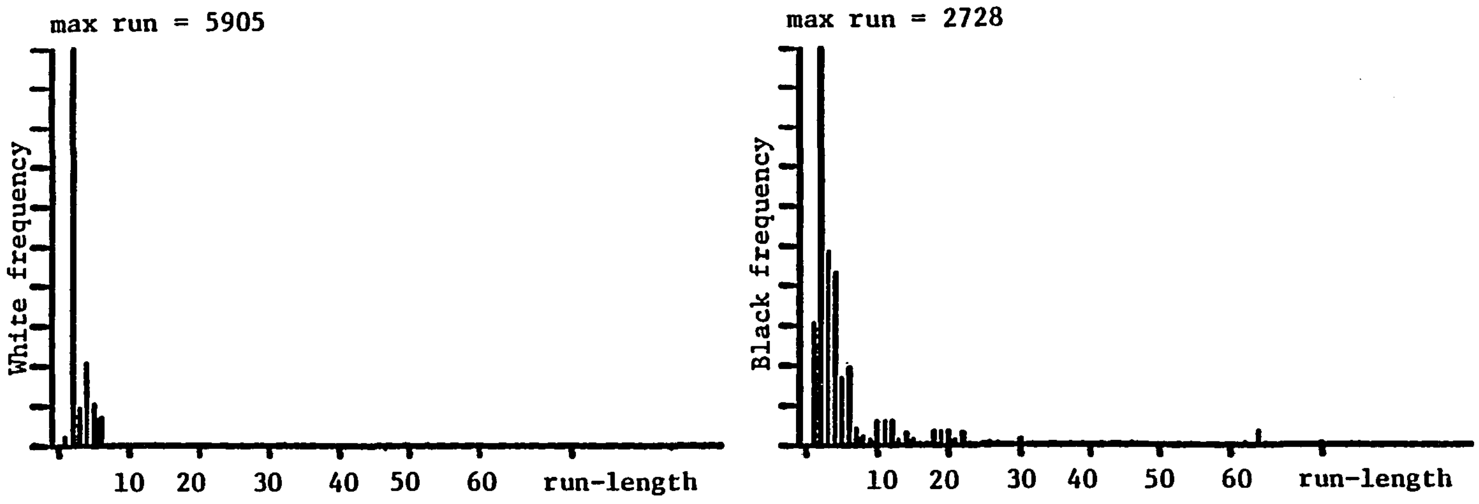
Figure 4.8. Frequency distribution of image "ecℓ1"



(a) White run-length distribution

(b) Black run-length distribution

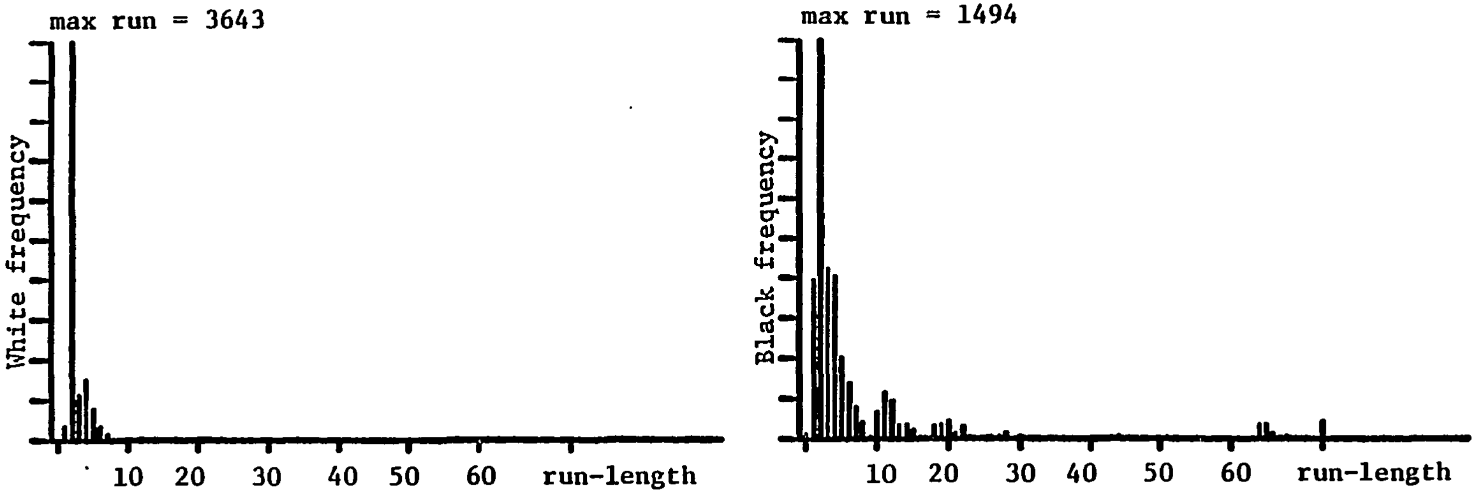
Figure 4.9. Frequency distribution of image "lotssin"



(a) White run-length distribution

(b) Black run-length distribution

Figure 4.10. Frequency distribution of image "doc4a"



(a) White run-length distribution

(b) Black run-length distribution

Figure 4.11. Frequency distribution of image "pagel"

where

H_d = entropy of the edge difference d(d = b1 - a1) in the vertical mode (state S_1)

$$= - \sum_{d=-3}^{3} P(d) \cdot \log_2 P(d) \qquad (4.18)$$

H_{1i} = entropy of the i run (i = 1 or 2) in the horizontal mode (state S_2)

$$= - \sum_{\ell_{ik}=0}^{73} P(\ell_{ik}) \cdot \log_2 P(\ell_{ik}) \qquad (4.19)$$

P(d) = the probability that (b1 - a1) is equal to d where d is an integer varying between -3 and 3

$P(\ell_i)$ = the probability that the i run (i=1 or 2) is equal to ℓ_i in the horizontal mode.

The average number of pels per state B_s is given by

$$B_s = P(S_1)\, r_{a0a1} + P(S_2)\, r_{a0b2} + P(S_3)(r_1 + r_2) \qquad (4.20)$$

where

r_{a0a1} = average of absolute value of a0a1, a0a1 = a1-a0

$$= \sum_{a0a1=1}^{640} P(a0a1) \cdot a0a1 \qquad (4.21)$$

r_{a0b2} = average value of pass mode distance a0b2

$$= \sum_{a0b2=2}^{640} P(a0b2) \cdot a0b2 \qquad (4.22)$$

r_1 = average length of first run in the horizontal mode

$$= \sum_{\ell_1=1}^{640} P(\ell_1) \cdot \ell_1 \qquad (4.23)$$

r_2 = average length of second run in the horizontal mode

$$= \sum_{\ell_2}^{640} P(\ell_2)\cdot\ell_2 \qquad (4.24)$$

The theoretical compression factor Q_{max} is calculated from the entropy per pel H_{pel} by the following formula:

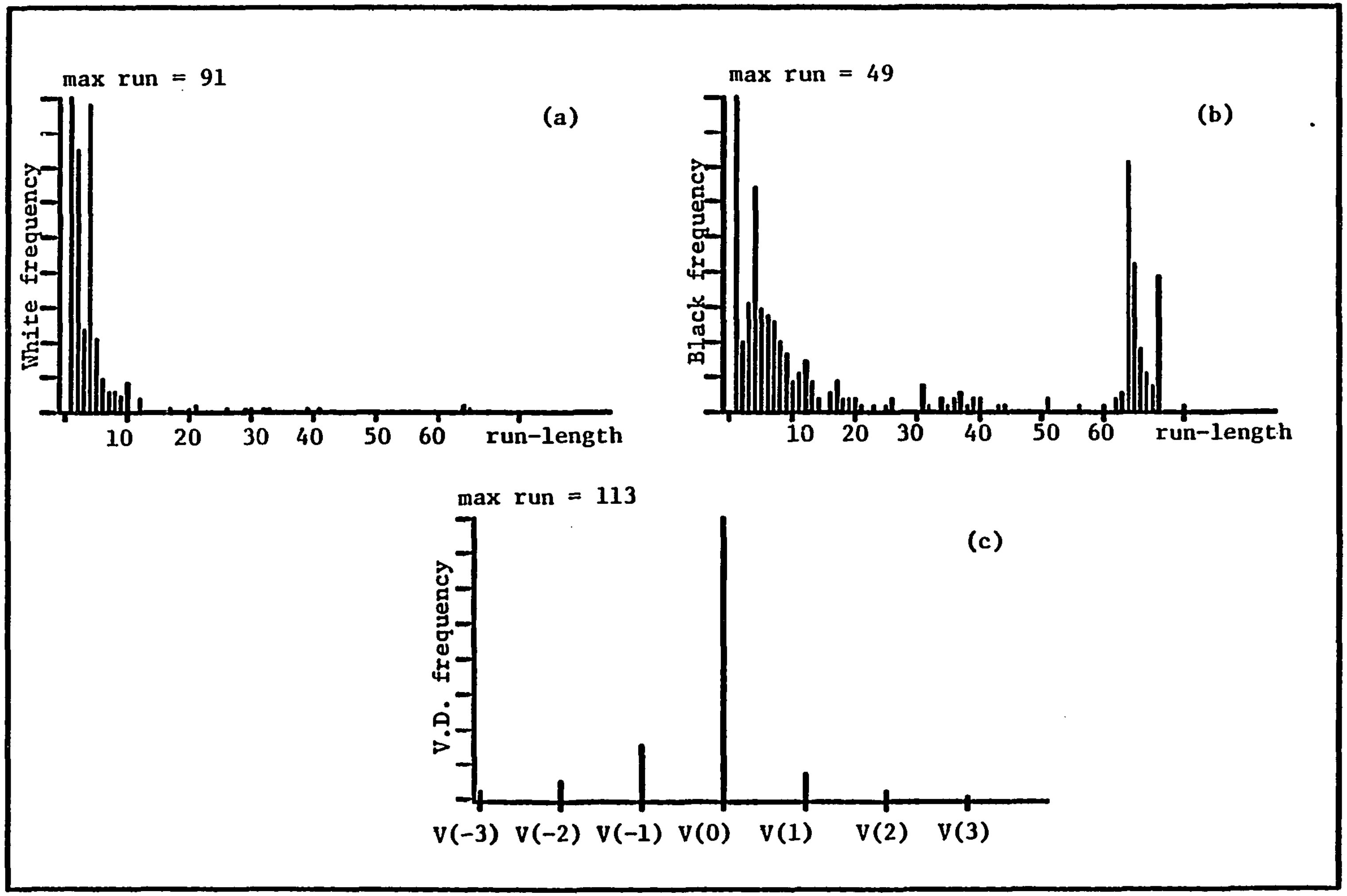
$$Q_{max} = \frac{1}{H_{pel}} \qquad (4.25)$$

Tables 4.17-4.24 include the theoretical compression factor, using the two dimensional model, for the images in the data base. Figures 4.12-4.21 show the distribution of the frequency of the run-lengths, in the horizontal mode, for a sample of images from the data base. Runs greater than 63 were broken into two runs as described by the standard. The figures also show the distribution of the vertical distance d.

4.7. Analysis of the Results

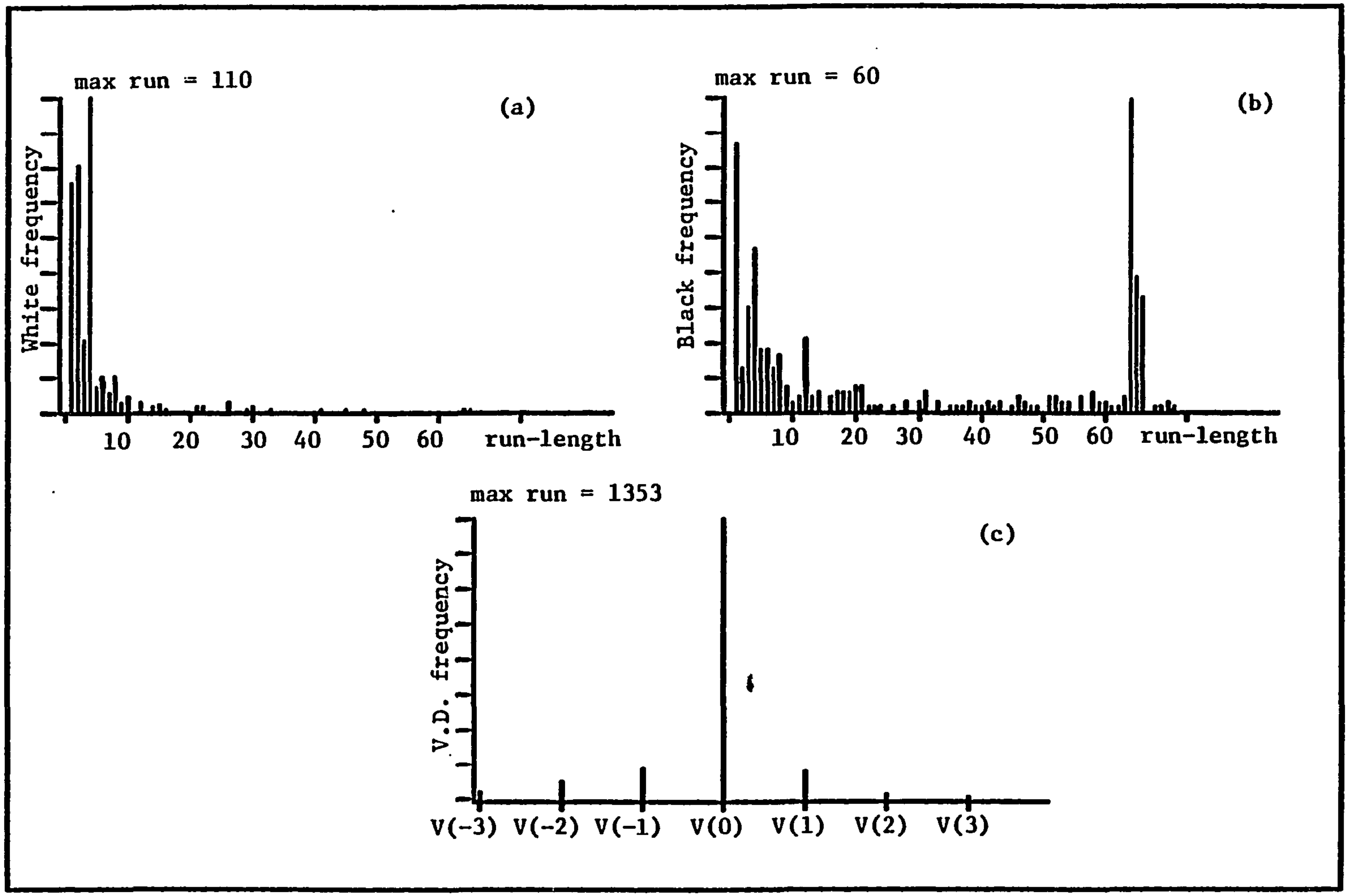
Looking at the results, we concluded the following points:

1) The two dimensional ($k = \infty$) coding technique gave better compression factor than the one dimensional coding technique except for the case of screens or blocks full of text. The ratio of the two dimensional compression factor to the one dimensional compression factor depended on the class of image to be compressed. In Table 4.25, the first three columns contain the compression factor averages of the pictures of each group calculated using one dimensional, two dimensional ($k = 2$), and two dimensional ($k = \infty$) techniques. This table shows that the ratio of the compression factors of the two dimensional ($k = \infty$) to



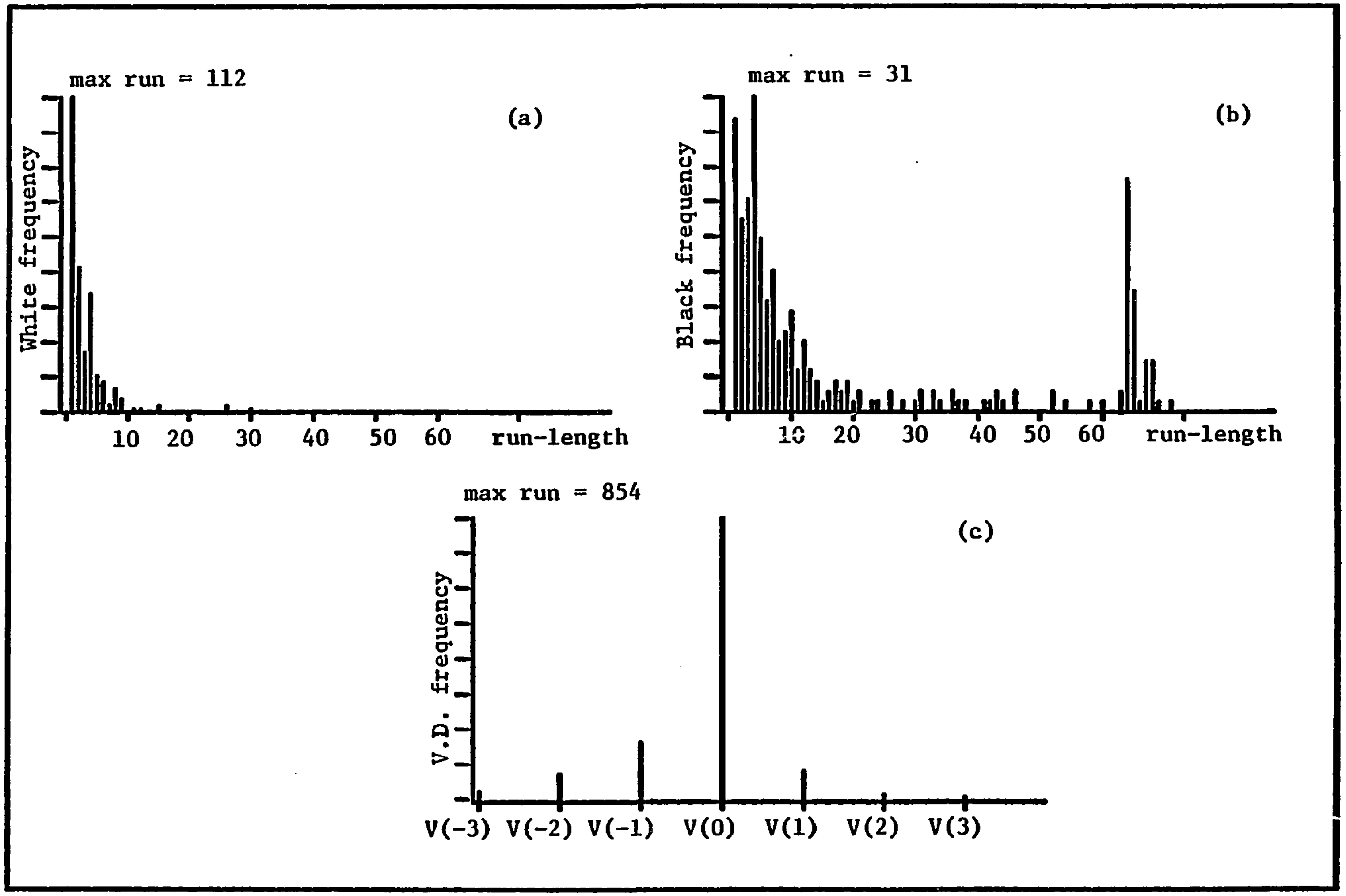
(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.12. Frequency distribution of image "doc2a"



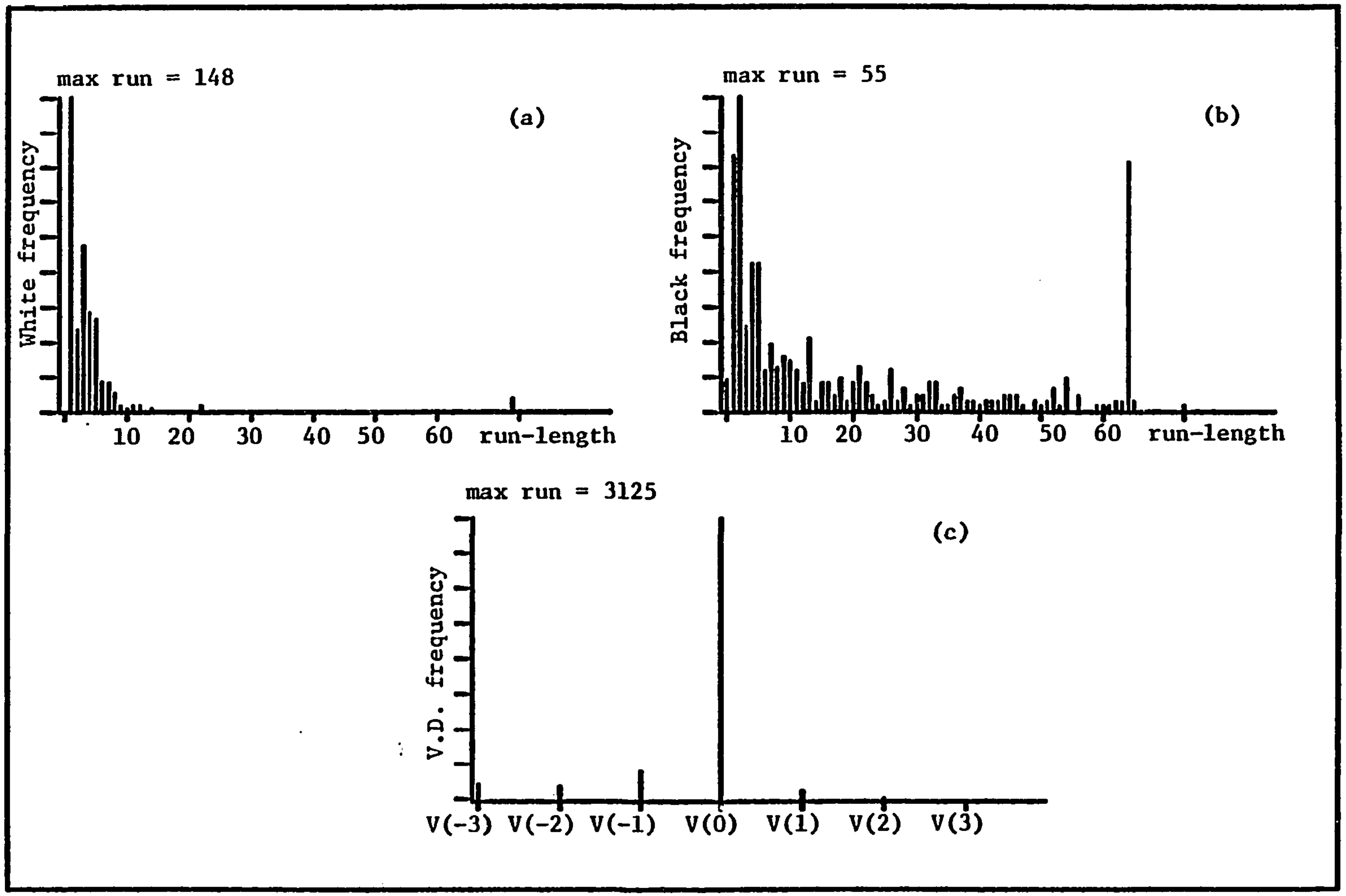
(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.13. Frequency distribution of image "doc2b"



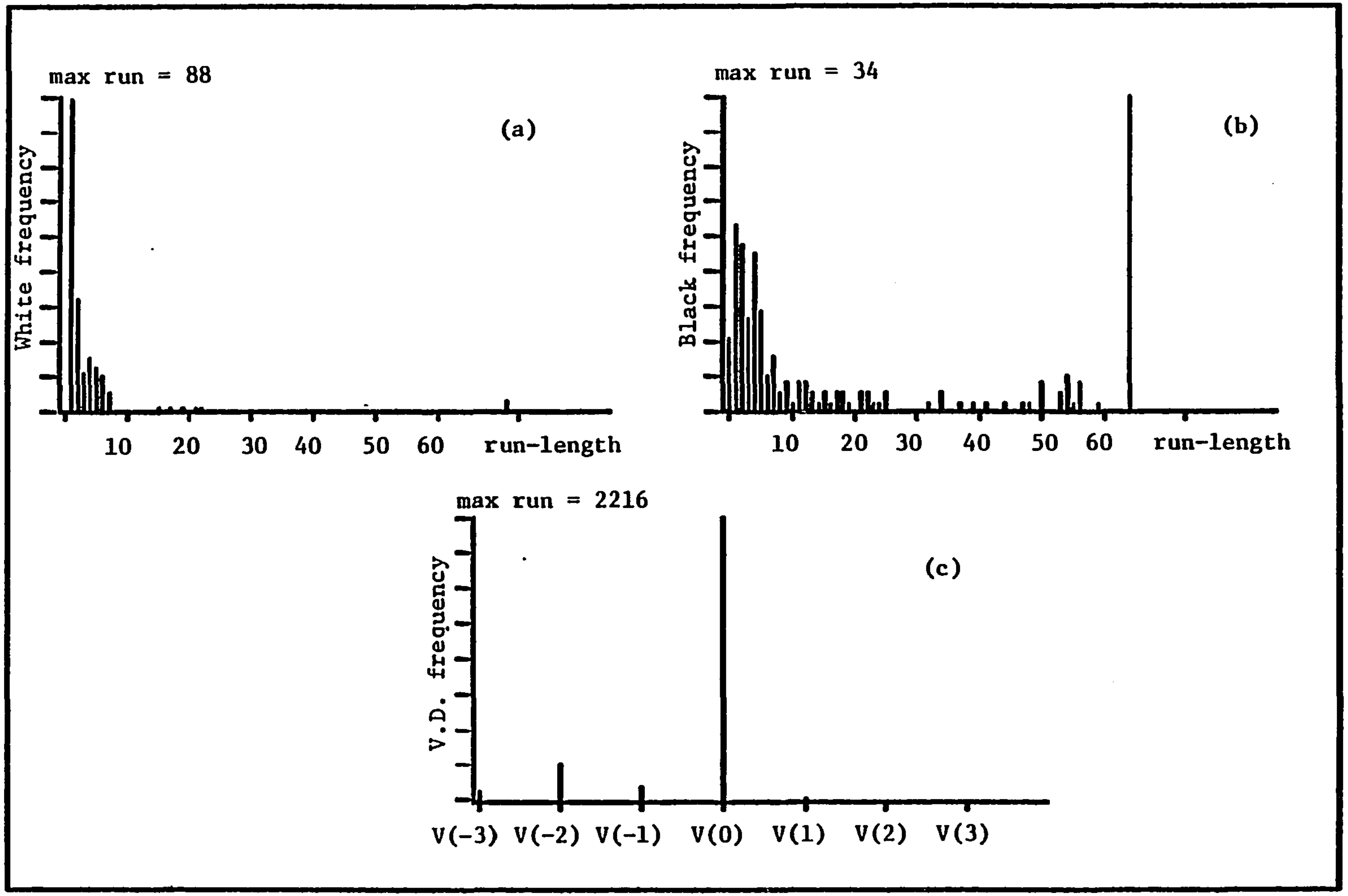
(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.14. Frequency distribution of image "doc2c"



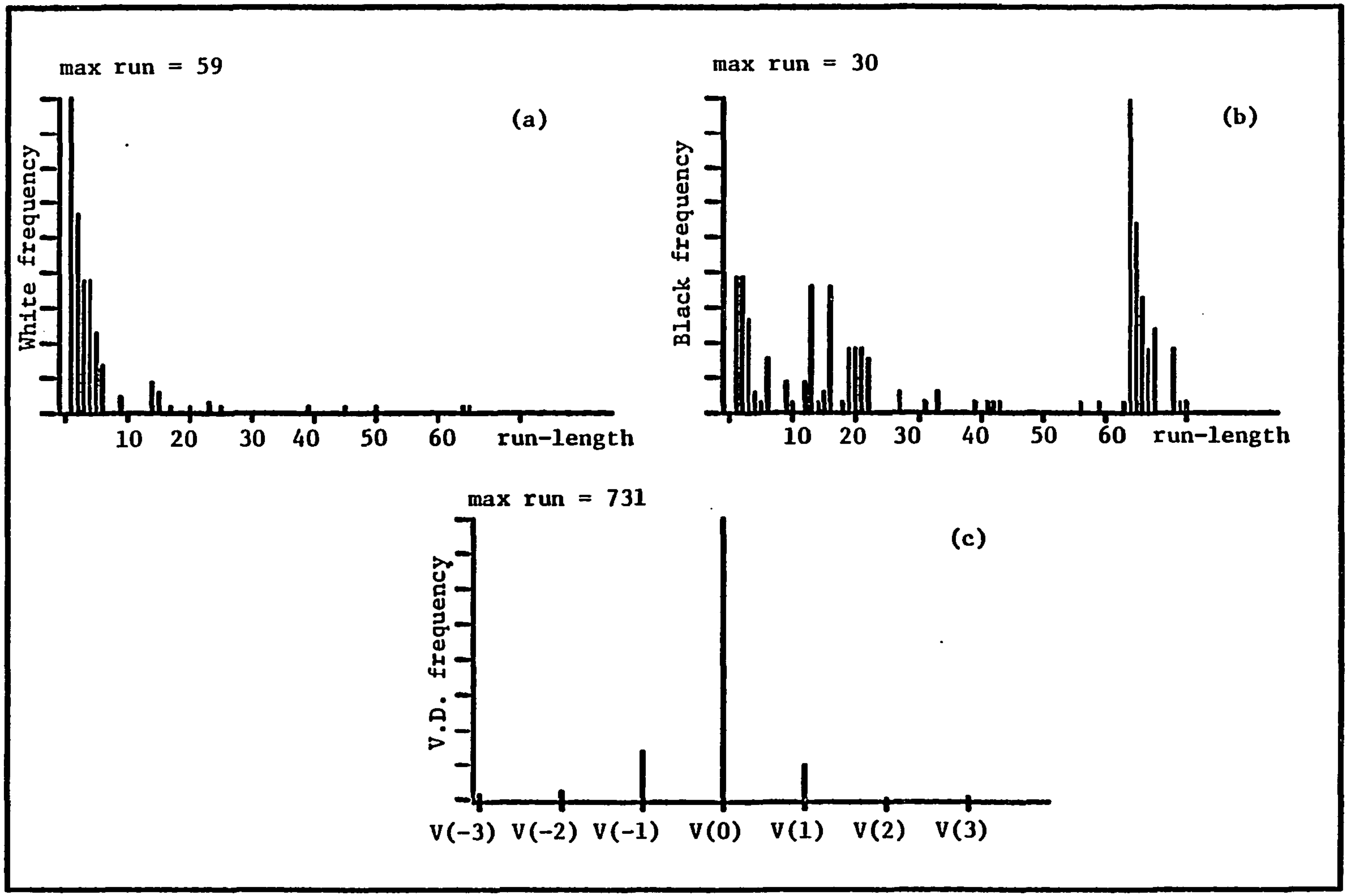
(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.15. Frequency distribution of image "doc6a"



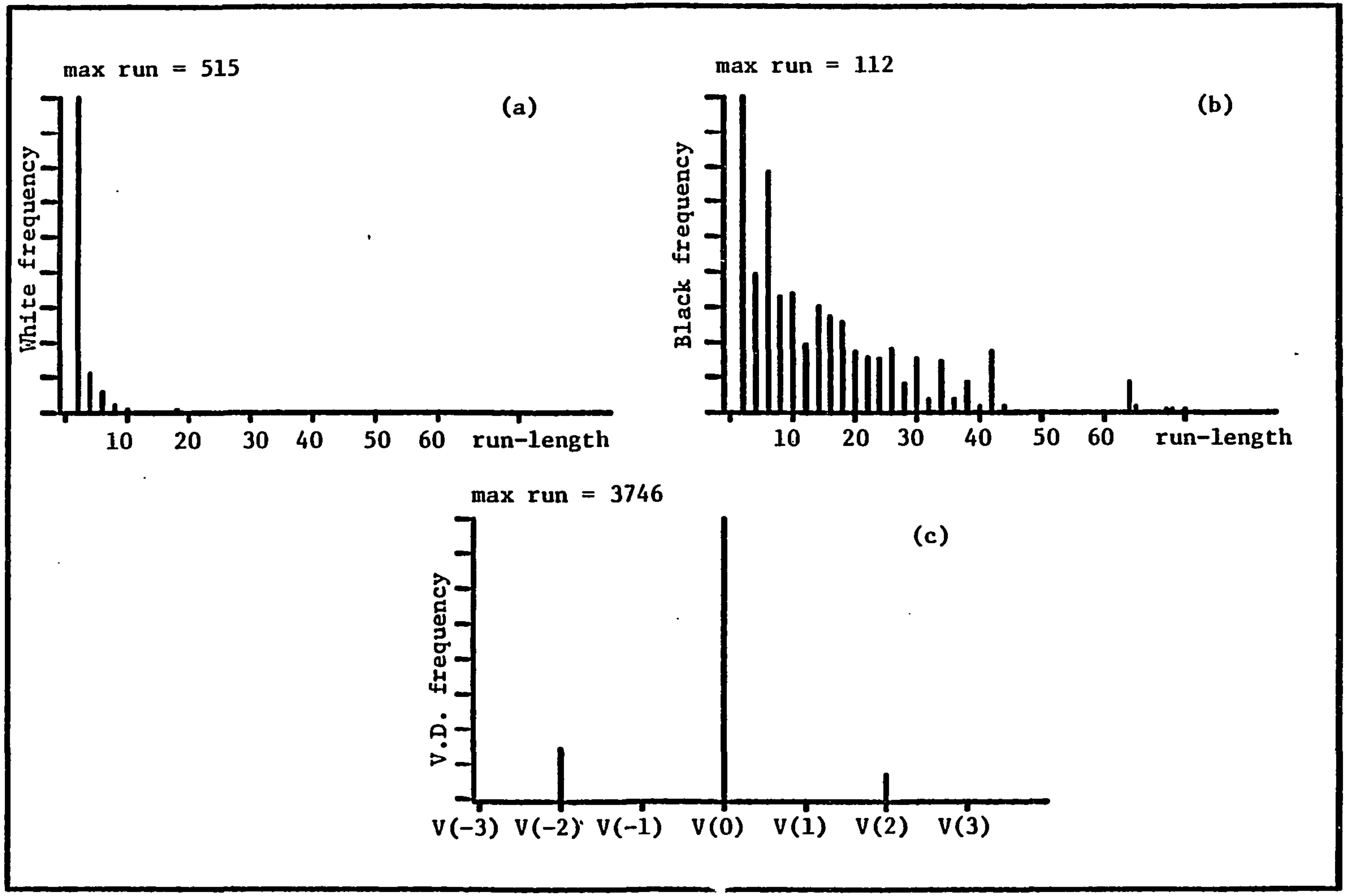
(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.16. Frequency distribution of image "doc6b"



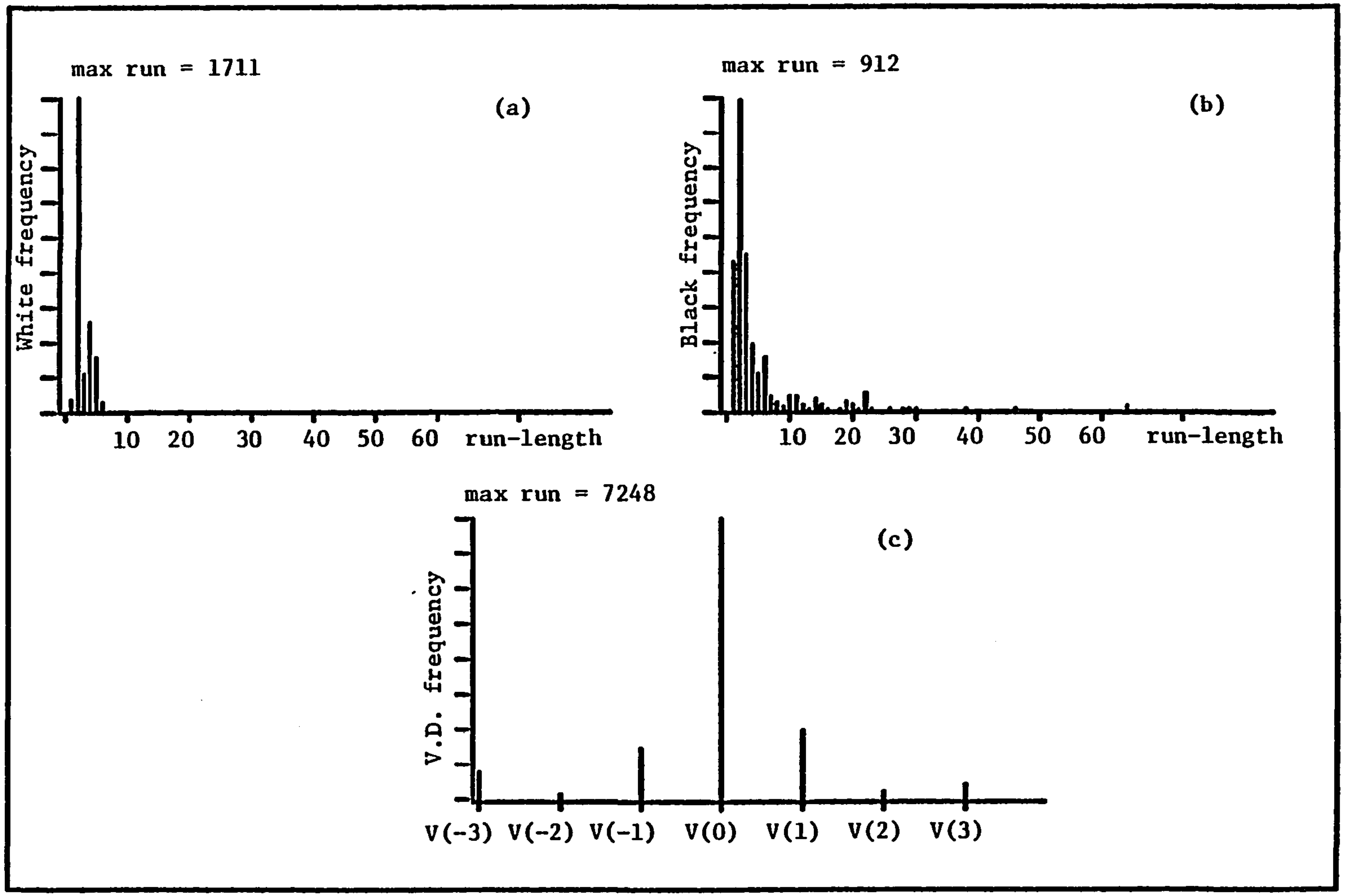
(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.17. Frequency distribution of image "ecℓ1"



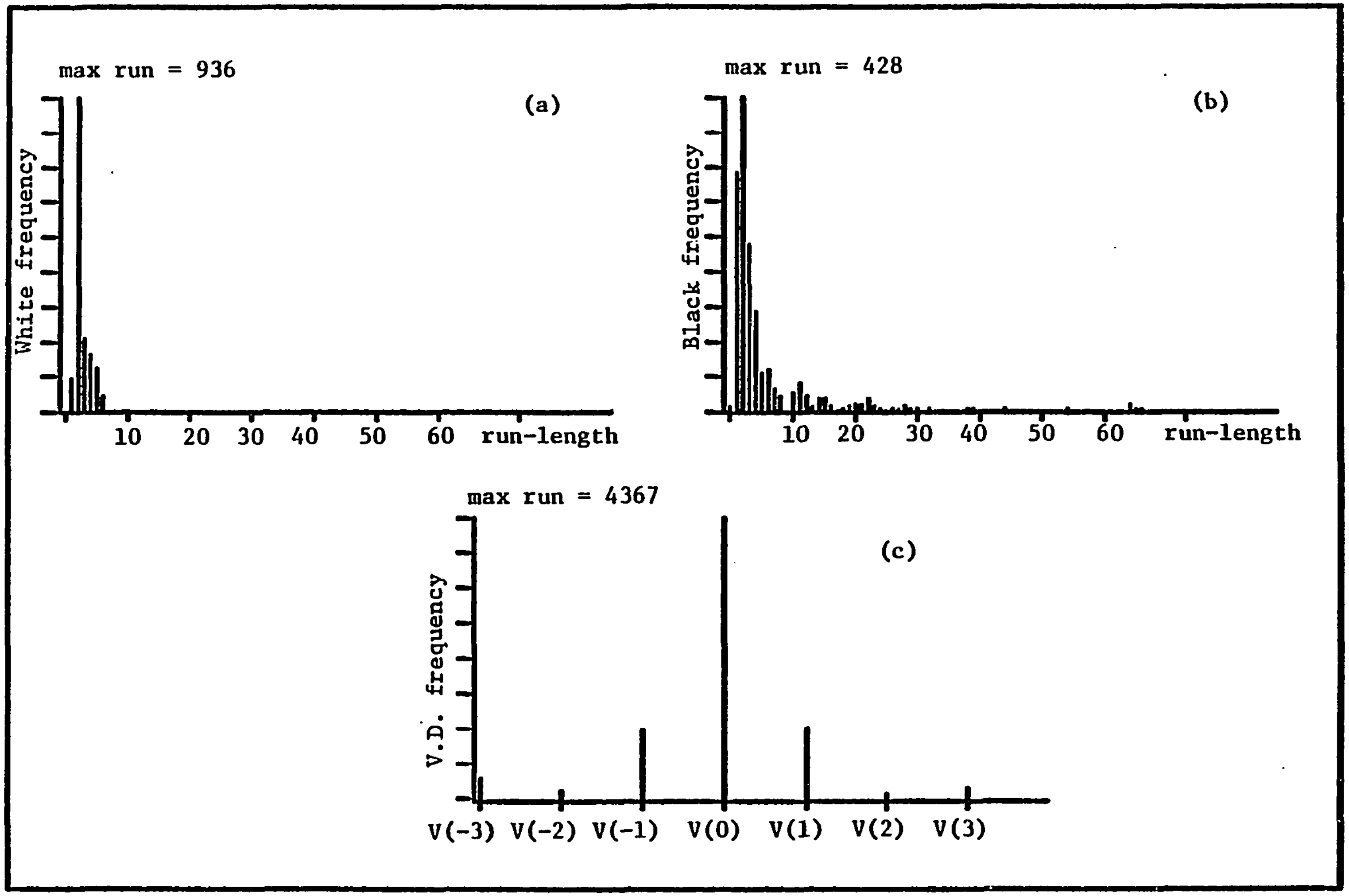
(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.18. Frequency distribution of image "lotssin"



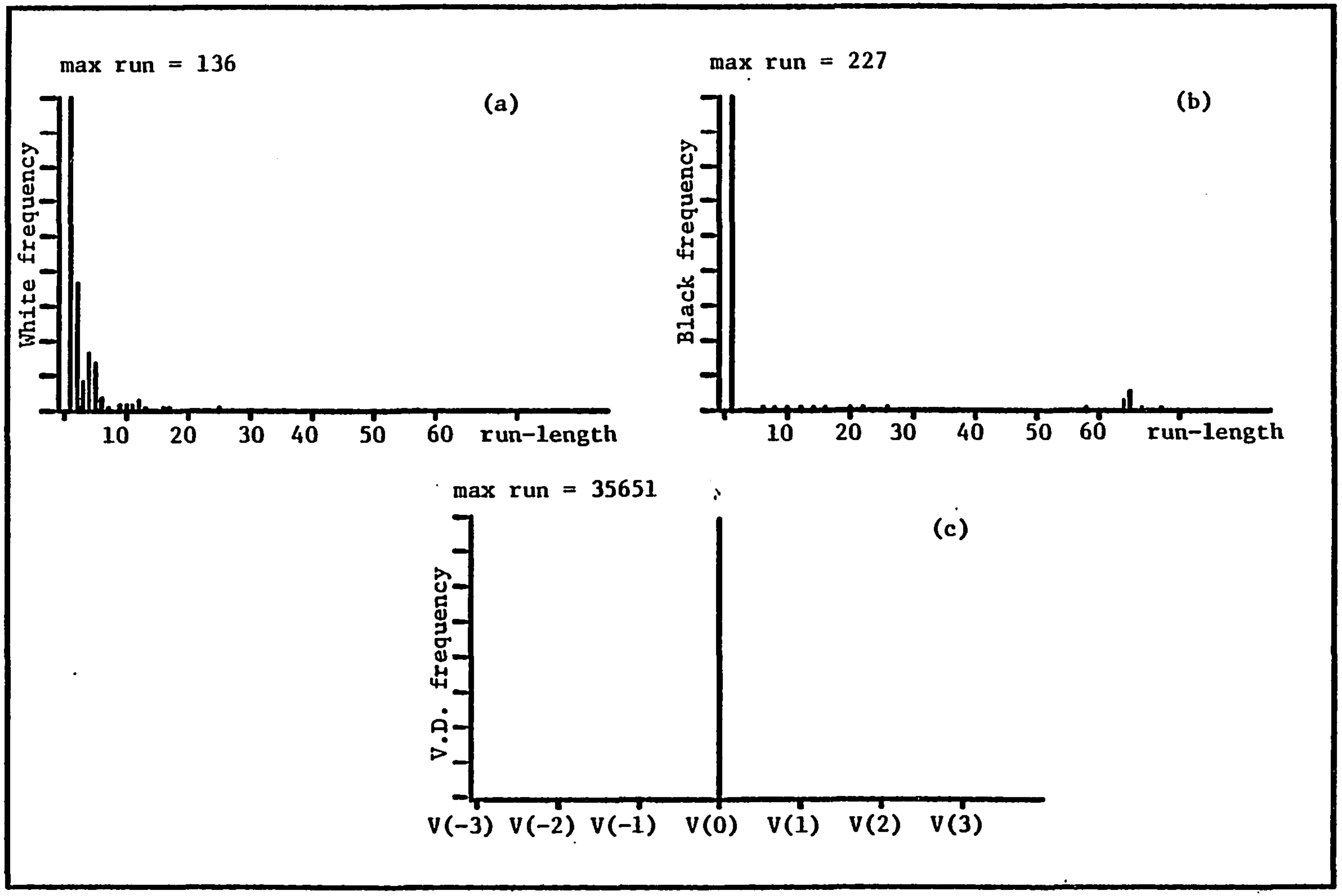
(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.19. Frequency distribution of image "doc4a"



(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.20. Frequency distribution of image "pagel"



(a) White run-length distribution (b) Black run-length distribution
(c) Vertical displacement (V.D.) distribution

Figure 4.21. Frequency distribution of image "usamap"

Table 4.25. Compression factors averages of each group of the image data base using each technique

Group #	1D[a]	2K[b]	FK[c]	2K/1D	FK/1D
Group 1	6.07	7.33	9.90	1.21	1.63
Group 2	5.52	6.54	8.38	1.18	1.52
Group 3	2.85	2.76	2.81	0.97	0.99
Group 4	3.54	3.70	3.97	1.05	1.12
Group 5	5.19	6.58	9.60	1.27	1.85
Group 6	2.69	3.07	3.70	1.14	1.38
AVERAGE	4.31	5.00	6.39	1.14	1.41

[a] 1D = compression factor using the CCITT one dimensional compression technique.

[b] 2K = compression factor using the CCITT two dimensional compression technique with k = 2.

[c] FK = compression factor using the CCITT two dimensional compression technique with $k = \infty$.

the one dimensional technique has an average of 1.41 with minimum and maximum equal to 0.99 and 1.85, respectively. The 0.99 ratio is for screens full of text and is the only ratio that is less than 1. The 1.85 ratio is for group 5 which consists of sample blocks of graphics.

2) In Table 4.25, it is shown that for screens full of graphics (group 2), the ratio of the average compression factor of the two dimensional, k = 2, to the average compression factor of the one dimensional is 1.18. This ratio is 1.52 for the case of two dimensional, $k = \infty$. This result shows that two dimensional technique with $k = \infty$ is the best choice for screens full of graphics.

3) From Table 4.25, it is clear that, for screens full of text (group 3), there is no significant difference between two dimensional and one dimensional compression factors. The average compression factor of the group using one dimensional technique is 2.85.

4) For screens that are a mixture of graphics and text blocks (group 4), Table 4.25 shows that the two dimensional compression factor is higher than one dimensional compression factor and the ratio of the average of the two dimensional to the average of the one dimensional compression factor is 1.05 and 1.12 for k = 2 and $k = \infty$, respectively. The one dimensional compression factor was found to have an average of 3.54.

5) In Table 4.25, it is shown that for blocks of graphics (group 5), the ratio of the average compression factor of the two dimensional technique, k = 2, to the average compression factor of the one dimen-

sional technique is 1.27. This ratio is 1.85 for the case of two dimensional technique, $k = \infty$. This result shows that the two dimensional technique with $k = \infty$ is the best choice for graphics blocks.

6) Screen pdraw3 contains 4 blocks. We compressed each of the 4 blocks separately and a big block containing all of these 4 blocks. The following two compression factors were calculated:

i) Compression factor of the big block = original size of the big block/size of the compressed block.

ii) Compression factor using the 4 small blocks to represent the big block = original size of the big block/ $\sum_{i=1}^{4}$ (size of the compressed block i).

Comparing the compression factors in i) and ii), we found that in ii) it is very slightly bigger than in i) using the one dimensional technique and almost the same when we used the two dimensional technique ($k = \infty$). Hence, it may be concluded from this example that dividing a big block into smaller blocks and compressing them individually will not give a better compression factor than in the case of compressing the big block as a whole. Besides, the division into smaller blocks will add more complexity and a small overhead of bytes that represents the sizes of the small blocks.

7) Table 4.26 contains the compression factors using the one dimensional and two dimensional ($k = \infty$) techniques taken from [13] for some CCITT standard documents. These values are normalized with reference to the compression factor of doc1 and included in the table. Table 4.27 contains similar values deduced from Tables 4.1 and 4.17. It

Table 4.26. Compression factors of the CCITT documents according to Reference [13]

Document #	1D[a]		2D[b]							
			Low[c]			High[d]				
	C.F.	Norm. C.F.[e]	Size	C.F.	Norm. C.F.	Size	C.F.	Norm. C.F.	Low/1D	High/1D
1	15.160	1.000	130684	15.709	1.000	175704	23.367	1.000	1.036	1.541
2	16.670	1.100	106851	19.212	1.223	117304	35.001	1.498	1.153	2.100
4	4.911	0.324	408261	5.028	0.320	585074	7.017	0.300	1.024	1.429
5	7.927	0.523	226285	9.072	0.578	288655	14.224	0.609	1.144	1.794
6	10.780	0.711	150572	13.634	0.868	164085	25.022	1.071	1.265	2.321
AVERAGE	11.090	0.732	204531	12.531	0.798	266164	20.926	0.896	1.124	1.837

[a] 1D = the CCITT one dimensional compression technique.

[b] 2D = the CCITT two dimensional compression technique with $k = \infty$.

[c] Low = document compressed in low resolution.

[d] High = document compressed in high resolution.

[e] Norm. C.F. = normalized compression factor.

Table 4.27. Compression factors of the CCITT documents using the CCITT one and two dimensional compression techniques

Document #	1D[a] C.F.	1D[a] Norm. C.F.[c]	2D[b] C.F.	2D[b] Norm. C.F.	2D C.F. / 1D C.F.
1	6.267	1.000	6.670	1.000	1.064
2	9.579	1.528	14.891	2.233	1.555
4	1.782	0.284	1.680	0.252	0.943
5	4.070	0.649	4.741	0.711	1.165
6	5.617	0.896	12.004	1.800	2.137

[a] 1D = the CCITT one dimensional compression technique.

[b] 2D = the CCITT two dimensional compression technique.

[c] Norm. C.F. = normalized compression factor.

was found that the ratios of two dimensional/one dimensional compression factors in the low resolution case in Table 4.26 were close to those in Table 4.27 except for doc2 and doc6. For the high resolution case, the only ratios that were close to each other in the two tables were those of doc6. This may be interpreted by noticing that low resolution mode was just enough to show the textural material in documents doc1, doc4, and doc5 which are documents that contain a lot of text. Similarly, the resolution of the screen was just enough to represent textual material in images doc1, doc4, and doc5.

8) To investigate the possibility of using a modified Huffman table with codes that are suitable to the screen statistics, frequency graphs for each image were generated. The coordinates of the horizontal axis in these figures represent the run-lengths while the coordinates of the vertical axis represent the number of times this run-length was used in compressing the picture. Runs greater than 63 were broken into two runs as described by the standard. From these graphs, we got the following remarks and conclusions:

a) Distribution of white runs has almost the same form in all the images. It has a concentration of small runs mostly located in the region between run 1 and run 6. The maximum run frequency occurs in run 1 for some of the images, specially graphics screens, and in run 2 for some other images, specially screens that have a lot of text. Since this maximum is not fixed, we might try to change the code so that, for the maximum frequency run, it varies with the image. We will show later that no big difference in compression factor can result from this change.

b) Frequency of the black runs is more distributed and varies from image to image with no fixed form. So, making variable code as suggested for the white runs in a) is not suitable. The frequency is also concentrated on small runs to the extent that the standard one dimensional code is efficient enough and no clear benefit can be seen from changing it.

Run length 1 has one of the highest frequencies, but the CCITT code assigns a code of length 6 while other less important frequencies are assigned a code of length 4. So, an improvement in the code may be found by assigning less bits to run-length 1.

c) Figures 4.3-4.5 show the distribution of the frequency for pictures doc2a, doc2b, and doc2c which represent graphics screens. Their distribution agrees with a) and b) above. Similar comments are applicable to doc6a, doc6b, and any graphics screen in groups 2, 5, and 7. Figures 4.6-4.9 show the frequency distribution for some graphics screens.

d) Figures 4.10 and 4.11 show the frequency distribution for pictures doc4a and page1 which represent screens full of text (group 3). The distribution of white is as explained in a) while the distribution of black is as in b) but more condensed than in graphics screens and more concentrated on small runs. To show that changing the code does not result in a big increase in the compression factor, we give the following example:

Table 4.3 shows that the compression factor of image doc4a is 1.96 which corresponds to a compressed image of size 8163 bytes. Figure 4.10 shows that white run-length 2 has a frequency equal to 5905. The modified Huffman table assigns a code of length 2 bits to this run. If a new code assigns 1 bit to this run (without going in details of this new code), the compressed buffer will decrease by 738 (= 5905/8) bytes. Hence, the new compressed size will be 2.15 (= 16000/(8163-738)). This represents 8% increase in the compression factor. Note that this calculation assumed that a code of length 1 bit was possible and neglected the negative effects of changing other codes in the table. In spite of that, the increase in compression factor is only 8%.

e) A calculation similar to the one in d) was done for doc6a, which is a sample of graphics screen, and showed 6% increase in the compression factor if the code was changed. Hence, we reached the same conclusion we got in d).

9) Comparing the compression factor of the two dimensional (k = ∞) coding technique with the theoretical compression factor of the one dimensional technique, we found that the former one was higher than the latter one except for documents containing a lot of text. So, two dimensional technique is the best choice.

10) Tables 4.1-4.8 show that, using the one dimensional technique, the average ratio of the real compression factor to the theoretical one is slightly low (0.68) for graphics screens and almost acceptable (0.77) for screens full of text. This result may be explained by two reasons. First, the code was optimized for the frequency of the runs in textual materials, but not for the frequency in graphics materials where it is hard to predict this frequency. Second, the model is not accurate for graphics screens because it assumed that black and white runs were independent of each other.

11) Tables 4.18-4.19 show that, using the two dimensional technique, the average ratio of the real compression factor to the theoretical one is 0.75 for graphics screens and 0.85 for screens full of text. Although the different variables that were used in calculating these compression factors were examined, no clear interpretation can justify why the model worked better in the case of screens full of text than in the case of graphics screens. The code of the first and second runs in the horizontal mode should not be considered as a part of the interpretation, as was the code for the runs in the one dimensional case, because the probability of the horizontal mode is almost the same in the two groups.

12) The average probabilities of the vertical, pass, and horizontal modes were found to be 0.75, 0.1, and 0.15, respectively. These are different from the values reported in [9] where the probability of the vertical mode was almost 0.9. This shows that the distributions of black and white pels in the computer screen are different than the same distributions in regular papers such as the CCITT standard documents.

13) The vertical mode was dominated by V(0). This fact and the result of the previous point indicate that the two dimensional technique worked as designed and to its limit.

14) The compression factor of the image usamap using the one dimensional technique was found to be less than 1. It was found to be 1.56 when using the two dimensional technique. The fact that these compression factors are low, even though the image usamap contains a lot of redundancy, indicates that these two techniques are not efficient for certain classes of images. Some examples of these classes are images that contain some repeated similar blocks or cross hatching. To overcome the deficiency found when the compression factor is less than 1, the standard techniques allow for uncompressed mode.

4.8. Conclusion

From the above analysis, we conclude that the CCITT standard two dimensional coding technique have better compression factor than the one dimensional technique, hence, should be our choice although its decompression time is higher. We also conclude that the two methods worked to their limit and produce satisfactory compression factors for the screen resolution. The facts that no improvement could be seen to changing the modified Huffman table and that for some class of data the two techniques are not efficient enough indicate we should search for other techniques.

5. APPLICATION OF THE LEMPEL-ZIV-WELCH ALGORITHM

In this chapter, we will present the LZW algorithm and the results of compressing the images of the data base defined in Chapter 3 using the LZW algorithm.

As a new major contribution, this research modifies the regular use of LZW in three different ways. The new modifications will be called LZWB, LZWB1, LZWB2-A, and LXWB2-B methods.

5.1. Description of the Lempel-Ziv-Welch Algorithm

The Lempel-Ziv-Welch (LZW) algorithm examines the data serially as a sequence of characters. It has a table to which it adds new strings of characters that it did not encounter before. Each entry "w,k" in the table consists of the symbol of a previously encountered string, w, and a character symbol, k. At each step, the algorithm searches for the string "w,k" in the table. If the string is found in the table, w is assigned the symbol of the string "w,k", k is assigned the value of the next input character, and a new search starts. If the string "w,k" is not found in the table, the symbol w is sent to the output, w is equated to k, k is equated to the next input character, and a new search starts. By this technique, the algorithm codes the input data according to its repeated strings and their distribution.

The first 256 symbols of the table are initialized to 256 characters, where each symbol content is equal to the symbol number. The string w and the character k are initialized to the values of the first and second characters in the input data, respectively. The size of the table is chosen to be 4096 symbols, so each symbol is represented by 12 bits. For more details, refer to [41] or Appendix D which has the listing of the code that simulates the LZW algorithm.

5.2. Method LZWB

The LZW algorithm compresses the data without any previous knowledge of its source. This may not be efficient enough when the source and some of its characteristics are known in advance. For the data this research works on, screens of text and graphics, the distribution of the black and white runs are known in advance. So, to let LZW benefit from this previously known source information, this research introduces a new solution that we call method LZWB. The proposed solution is to count the black and white runs in the image and then send the codes of these runs to LZW for compression. The letter "B" in method LZWB stands for "binary".

Method LZWB assumes that the first 128 symbols in the LZW Table represent run-lengths 1 to 128 of black pels and the symbols 129 to 256 represent run-lengths 1 to 128 of white pels. The input first goes through a counter which counts runs between 1 and 128. Any run-length greater than 128 is divided into one or more multiples of 128 and a run-length smaller than 128. The output of this counter is fed to the LZW algorithm for compression. The output of the counter may be greater than the size of the original block in some cases but it is expected that the distribution of the runs makes this data more suitable to compression than the original data. This better compressibility comes from the facts that certain runs are more frequent than the others and the dependency among the different runs is present in the form of repeated strings. Appendix E gives the code necessary to simulate the LZWB method.

5.3. Method LZWB1

Method LZWB1, as proposed by this research, assumes the first 200 characters in the LZW table to represent run-lengths 1 to 100 of black and white pels. The remaining 56 symbols of characters in the table are used to represent two or three consecutive run-lengths. Table 5.1 has these runs and their corresponding symbols. These runs were chosen because their probabilities, as given in the CCITT modified Huffman table, are the highest among other runs.

Table 5.2 shows the most probable black and white run-lengths and the lengths of their corresponding codes as defined in the modified Huffman table. The Huffman table is optimum if the probabilities of the entries are in the form $(1/2)^n$ where n is an integer greater than or equal to 1. We assume that the table is optimum and, hence, calculate the probabilities as given in Table 5.2. According to Table 5.2, white run lengths 1 to 4 have a total probability equal to 75% of the white run-lengths whereas black run-lengths 2 to 7 have (6/16) of the black run-lengths. So, from the white run-lengths, we only used run-lengths 1 to 4 in the symbols. As for the black run-lengths, we chose run-

Table 5.1. The probability and code length of some run-lengths derived from Table I in [13]

Black runs			White runs		
Run length	Code length	Run prob.	Run length	Code length	Run prob.
2	4	1/16	2	2	1/4
3	4	1/16	3	2	1/4
4	4	1/16	1	3	1/8
5	4	1/16	4	3	1/8
6	4	1/16			
7	4	1/16			
8	5	1/32			
9	5	1/32			
10	5	1/32			
11	5	1/32			
64	5	1/32			
128	5	1/32			

Table 5.2. The strings used in LZWB1 and their corresponding symbols

Symbol	String starting with black pel	Symbol	String starting with white pel
200	01	212	10
201	001	213	110
202	0001	214	1110
203	00001	215	11110
204	011	216	100
205	0011	217	1100
206	00011	218	11100
207	000011	219	111100
208	0111	220	1000
209	00111	221	11000
210	000111	222	111000
211	0000111	223	1111000
224	010	240	101
225	0100	241	1011
226	01000	242	10111
227	010000	243	101111
228	0010	244	1101
229	00100	245	11011
230	001000	246	110111
231	0010000	247	1101111
232	0110	248	1001
233	00110	249	11001
234	01100	250	10011
235	001100	251	110011
236	01110	252	10001
237	001110	253	110001
238	011100	254	100011
239	0011100	255	1100011

lengths 1 to 4; we did not choose run-lengths 5 to 7 because we wanted to simplify the operation although if there is a benefit or not of including them is a point that needs more research. The CCITT modified Huffman table assumes that the frequency of black run-length 1 is smaller than the probability of any run-length between 2 and 11, but our distribution analysis in Chapter 4 showed that frequency of black run-length 1 was comparable to that of run length 3 and might be a little less than run length 2. So, in the symbols we chose, we also represented run-length 1.

Method LZWB1 is a step beyond LZWB and, as in method LZWB, we predict that the output of the counter is more compressible than the original data. We also predict that, since some of the symbols represented two or three of the most frequent runs, the size of the counter output will not be as big as the size of the counter output in LZWB. Appendix F gives the code necessary to simulate method LZWB1.

5.4. Method LZWB2

The LZW algorithm initializes the first 256 symbols to character symbols. Since it has no previous knowledge of the symbols in the input data, it does not try to initialize symbols other than the characters symbols. The symbols of method LZWB2, as in LZWB, represent white and black run-lengths; hence, we assume that LZWB2 has a prior knowledge of the frequency of the symbols and benefit from this knowledge by initializing some symbols, from symbol 257 and above, to symbols of strings that are very likely to occur.

Two table initializations were tried. The symbols and their corresponding run-lengths for these two initializations are presented in Tables 5.3 and 22.1. The initialization of the table requires a change in the code of the LZW decompression process. The change needed is to allow for the first received symbol to be a string symbol in the form of "w,k". The code of this is inserted before the code of the decompression used in LZW. Appendix G gives the code necessary to simulate method LZWB2 using Table 5.3. We will call this combination method LZWB2-A. The code of method LZWB2 using Table 22.1 is exactly the same as the code using Table 5.3 except for the part of initializing the table which differs by the number of symbols to be initialized. We will call method LZWB with the LZW table initialized by Table 22.1 as method LXWB2-B.

5.5. Results of LZW and the Above Mentioned Modifications

The results of compressing the images in the data base using the LZW algorithm are presented in Tables 5.4-5.10. Tables 5.11-5.15 give the results of the average values for each group when compressed by methods LZW, LZWB, LZWB1, LZWB2-A, and LZWB2-B. Note, that for the methods LZWB, LZWB2-A, and LZWB2-B, the results for group 8 do not include the image "usamap" because the result of the symbols counter is bigger than the buffer used. In the following sections, we will try to analyze the above results.

Table 5.3. Extended LZW tables to be used with LZWB2-A

Symbol	String	w	k
256	01	0	128
257	001	1	128
258	0001	2	128
259	00001	3	128
260	011	0	129
261	0011	1	129
262	00011	2	129
263	000011	3	129
264	0111	0	130
265	00111	1	130
266	000111	2	130
267	0000111	3	130
268	010	256	0
269	0100	256	1
270	01000	256	2
271	010000	256	3
272	0010	257	0
273	00100	257	1
274	001000	257	2
275	0010000	257	3
276	0110	260	0
277	00110	261	0
278	01100	260	1
279	001100	261	1
280	01110	264	0
281	001110	265	0
282	011100	264	1
283	0011100	265	1
284	10	128	0
285	110	129	0
286	1110	130	0
287	11110	131	0
288	100	128	1
289	1100	129	1
290	11100	130	1
291	111100	131	1
292	1000	128	2
293	11000	129	2
294	111000	130	2
295	1111000	131	2
296	101	284	128
297	1011	284	129

Table 5.3. continued

Symbol	String	w	k
298	10111	284	130
299	101111	284	131
300	1101	285	128
301	11011	285	129
302	110111	285	130
303	1101111	285	131
304	1001	288	128
305	11001	289	128
306	10011	288	129
307	110011	289	129
308	10001	292	128
309	110001	293	128
310	100011	292	129
311	1100011	293	129

Table 5.4. Results of compressing images in group 1 using method LZW

Image	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
doc1a	6.62	19.93	1.59	1867	0
doc1b	3.69	19.55	1.53	3142	0
doc1c	13.25	12.91	1.60	1060	0
doc2a	7.71	11.21	1.54	1638	0
doc2b	6.82	14.55	1.59	1818	0
doc2c	10.01	14.99	1.60	1321	0
doc4a	2.91	33.67	1.54	3917	0
doc4b	2.55	32.24	1.54	4096	341
doc4c	2.31	8.74	0.66	2283	0
doc51a	3.88	18.62	1.21	2454	0
doc51b	5.44	11.75	1.27	1822	0
doc51c	8.41	5.28	0.71	838	0
doc5ra	2.97	15.87	1.16	2946	0
doc5rb	5.87	11.48	1.16	1617	0
doc5rc	3.18	6.54	0.66	1703	0
doc6a	4.86	18.84	1.59	2448	0
doc6b	6.81	21.58	1.60	1822	0
doc8	5.77	16.75	1.60	2104	0
AVERAGE	5.73	16.36	1.34	2161	19

Table 5.5. Results of compressing images in group 2 using method LZW

Image	Comprs. factor	Comprs. time s	Decmprs. time s	Table size	Extra calls
frnch3a	5.63	16.31	1.60	2148	0
flowchrt	4.57	18.18	1.59	2589	0
electrc	3.87	22.90	1.53	3012	0
ordrfrm	5.17	17.91	1.53	2316	0
frnch1a	6.11	17.14	1.53	2001	0
doc2a	7.71	11.15	1.59	1638	0
doc2b	6.82	14.50	1.59	1818	0
AVERAGE	5.70	16.87	1.57	2217	0

Table 5.6. Results of compressing images in group 3 using method LZW

Image	Comprs. factor	Comprs. time s	Decmprs. time s	Table size	Extra calls
romtxt	2.35	34.43	1.54	4096	691
frnch2a	2.91	27.91	1.54	3925	0
page1	4.15	17.75	1.59	2825	0
doc1-2	4.68	18.78	1.59	2535	0
cprog	7.07	17.19	1.54	1763	0
doc1b	3.69	19.56	1.54	3142	0
doc4a	2.91	33.67	1.53	3917	0
doc4b	2.55	32.24	1.54	4096	341
AVERAGE	3.79	25.19	1.55	3287	129

Table 5.7. Results of compressing images in group 4 using method LZW

Image	Comprs. factor	Comprs. time s	Decmprs. time s	Table size	Extra calls
pdraw3	4.58	25.43	1.59	2585	0
science1	3.53	24.50	1.54	3279	0
science2	2.77	29.44	1.54	4096	10
doc51a	3.88	18.62	1.21	2454	0
AVERAGE	3.69	24.50	1.47	3104	3

Table 5.8. Results of compressing images in group 5 using method LZW

Image	Comprs. factor	Comprs. time s	Decmprs. time s	Table size	Extra calls
opamp1	5.94	7.09	0.93	1326	0
opamp2	6.07	13.46	1.54	1934	0
ecl1	7.77	6.54	0.94	1078	0
ecl2	7.98	10.27	1.43	1423	0
netwrk	6.16	10.77	1.32	17.28	0
table1	3.47	14.01	1.05	2331	0
usa1	9.13	5.11	0.82	852	0
doc51a	4.79	2.31	0.33	785	0
doc5rb	5.03	5.22	0.71	1198	0
lotssin	3.56	10.66	0.99	2087	0
frnch3b	5.11	4.39	0.54	1006	0
barchrt	5.27	3.79	0.49	909	0
barchrt	3.36	0.82	0.16	518	0
barchrt	5.58	1.70	0.27	609	0
test2	4.03	3.03	0.44	992	0
test3	3.43	3.19	0.44	1121	0
test4	3.21	3.35	0.44	1184	0
test5	3.04	3.95	0.44	1324	0
diag1	4.49	3.07	0.44	932	0
diag2	4.78	1.92	0.27	666	0
daig3	3.81	2.30	0.33	853	0
diag4	3.58	2.03	0.28	842	0
diag5	7.63	2.86	0.49	683	0
diag5s	5.41	1.38	0.27	552	0
diag6	5.27	1.16	0.17	513	0
diag6	4.71	0.88	0.16	479	0
diag6	8.62	4.01	0.61	749	0
netwrk2	3.36	1.65	0.28	731	0
pdraw1	3.19	1.92	0.28	864	0
usa2	2.06	0.32	0.06	441	0
usa2	4.01	1.20	0.22	609	0
doc51b	5.62	0.99	0.17	474	0
science3	2.54	1.32	0.16	746	0
science3	1.85	0.33	0.05	453	0
AVERAGE	4.82	4.03	0.52	970	0

Table 5.9. Results of compressing images in group 7 using method LZW

Image	Comprs. factor	Comprs. time s	Decmprs. time s	Table size	Extra calls
bignames	2.11	37.68	1.54	4096	1222
sun	2.95	27.73	1.54	3872	0
hazard	2.66	28.73	1.54	4096	166
mansc1	2.35	34.93	1.53	4096	690
mansc2	3.31	23.01	1.54	3478	0
fig2	8.42	14.17	1.60	1522	0
fig4	7.86	12.85	1.53	1612	0
fig6	3.69	22.74	1.54	3149	0
fig7	5.09	15.98	1.53	2350	0
fig8	3.43	23.84	1.54	3364	0
AVERAGE	4.19	24.17	1.54	3164	208

Table 5.10. Results of compressing images in group 8 using method LZW

Image	Comprs. factor	Comprs. time s	Decmprs. time s	Table size	Extra calls
blok3	27.97	10.49	1.59	636	0
blok6	10.98	14.06	1.59	1226	0
boxes	16.06	10.38	1.60	919	0
lines	15.19	12.25	1.59	957	0
test1	12.79	2.47	0.44	487	0
usamap	6.57	6.59	0.77	1089	0
AVERAGE	14.93	9.37	1.26	886	0

Table 5.11. Results of compressing each group of the image data base suing method LZW

Group #	Comprs. factor	C.F. / FAX	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	5.73	0.58	16.36	1.34	2161	19
GROUP 2	5.70	0.68	16.87	1.57	2217	0
GROUP 3	3.79	1.35	25.19	1.55	3287	129
GROUP 4	3.69	0.93	24.50	1.47	3104	3
GROUP 5	4.82	0.50	4.03	0.52	970	0
GROUP 7	4.19	1.13	24.17	1.54	3164	208
AVERAGE	4.65	0.86	18.52	1.33	2484	60

Table 5.12. Results of compressing each group of the image data base using method LZWB

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Cmprs. time s	Dcmprs. time s	Count smbl.	Table size	Extra calls
GROUP 1	5.88	0.59	1.02	13.80	1.03	6425	2227	164
GROUP 2	5.42	0.65	0.95	13.11	1.03	6428	2368	0
GROUP 3	2.96	1.05	0.78	26.75	2.39	14931	3588	785
GROUP 4	3.37	0.85	0.91	20.47	1.42	8853	3284	44
GROUP 5	5.18	0.54	1.07	2.83	0.29	1768	970	0
GROUP 7	3.79	1.02	0.90	26.95	1.99	12218	3306	362
AVERAGE	4.43	0.78	0.94	17.32	1.36	8437	2624	226

Table 5.13. Results of compressing each group of the image data base using method LZWB1

Group #	Cmprs. factor	C.F. FAX	C.F. LZW	Cmprs. time s	Dcmprs. time s	Count smbl.	Table size	Extra calls
GROUP 1	5.86	0.59	1.02	12.31	0.91	5057	2224	158
GROUP 2	5.41	0.65	0.95	11.86	0.94	4973	2369	0
GROUP 3	3.01	1.07	0.79	29.00	1.96	9994	3566	751
GROUP 4	3.37	0.85	0.91	18.38	1.21	6297	3285	43
GROUP 5	5.19	0.54	1.08	2.61	0.28	1491	958	0
GROUP 7	3.89	1.05	0.93	23.77	1.66	8584	3285	371
AVERAGE	4.46	0.79	0.95	16.32	1.16	6066	2615	221

Table 5.14. Results of compressing each group of the image data base using method LZWB2-A

Group #	Cmprs. factor	C.F. FAX	C.F. LZW	Cmprs. time s	Dcmprs. time s	Count smbl.	Table size	Extra calls
GROUP 1	6.00	0.61	1.05	13.72	1.02	6425	2255	168
GROUP 2	5.49	0.66	0.96	13.29	1.04	6428	2401	0
GROUP 3	2.98	1.06	0.79	27.04	2.39	14931	3608	803
GROUP 4	3.40	0.86	0.92	20.48	1.42	8853	3310	50
GROUP 5	5.29	0.55	1.10	2.91	0.30	1768	1012	0
GROUP 7	3.81	1.03	0.91	27.79	1.99	12218	3330	380
AVERAGE	4.50	0.79	0.95	17.54	1.36	8437	2653	234

Table 5.15. Results of compressing each group of the image data base using method LZWB2-B

Group #	Cmprs. factor	C.F. / FAX	C.F. / LZW	Cmprs. time s	Dcmprs. time s	Count smbl.	Table size	Extra calls
GROUP 1	6.28	0.63	1.10	15.77	1.02	6425	2493	196
GROUP 2	5.72	0.68	1.00	15.44	1.05	6428	2661	0
GROUP 3	3.07	1.09	0.81	30.79	2.39	14931	3731	925
GROUP 4	3.53	0.89	0.96	22.73	1.42	8853	3485	105
GROUP 5	5.48	0.57	1.14	3.63	0.29	1768	1323	0
GROUP 7	3.86	1.04	0.92	26.14	1.99	12218	3512	494
AVERAGE	4.66	0.82	0.99	19.08	1.36	8437	2868	287

Table 5.16. Compression and decompression times averages for each group when compressed by the CCITT two dimensional compression technique with k = ∞

Group #	Comprs. time s	Dcmprs. time s
GROUP 1	3.70	2.96
GROUP 2	4.09	3.22
GROUP 3	6.37	5.86
GROUP 4	4.63	3.90
GROUP 5	1.48	1.14
GROUP 7	5.58	4.88
AVERAGE	4.31	3.66

5.6. LZW vs. FAX

By method FAX here and throughout the rest of the thesis, we mean, unless otherwise specified, the CCITT two dimensional coding technique with $k = \infty$. The results of the average compression factor (c.f.) for each group were presented in Table 4.25 and the results of the compression and decompression times are presented in Table 5.16. Comparing the results in the above tables with the results in Table 5.11, we get the following points:

1) Compression factor: FAX gives higher c.f. than LZW for graphics data, such as group 2 (g2) and g5, and LZW gives higher c.f. than FAX for g3 and g7. This means when the data consist of mainly long black runs and short white runs FAX outperforms LZW, but when the data consists of mainly small runs, of black and white pels, LZW outperforms FAX. For the data that are mixed of short and long runs, it seems that FAX outperforms LZW as in group 4 or the average of group 1.

2) LZW needs longer compression time (c.t.), almost 4 times the time used by FAX. But the LXW decompression time (d.t.) is smaller than that of FAX, almost 0.36 times the time used by FAX. The decompression times are in the range of 3 s and 1 s for FAX and LZW, respectively.

5.7. LZWB and LZWB2 vs. LZW and FAX

From Tables 5.11 and 5.12, we observe that LZWB advantages over LZW are that groups 1 and 5 have higher c.f. and lower d.t. and c.t. than those of LZW. The disadvantages are that the overall c.f. is smaller and the table size is bigger. So, in general, LZW is still better than LZWB.

Tables 5.14 and 5.15 show that initializing the LZW table, as in LZWB2-A and LZWB2-B, gave slight improvement in the c.f. and the bigger the initialized part is the bigger the increase in c.f. is. The increase in the LZWB2-B c.f. were 10% and 14% over the c.f. of LZW for g1 and g5, respectively. These increases are 3% and 7% for LZWB. The d.t. are very small for g5, average of 29 s, with LZWB, LZWB2-A, and LZWB2-B. The disadvantages of the initialization are that the c.t. and the counter output increase slightly with the initialized portion.

Compared to FAX, methods LZWB, LZWB2-A, and LZWB2-B have c.f. no more than 10% higher for g3 and g7. But the c.f. of LZWB, LXWB2-A, and LZWB2-B are less than the c.f. of LZW for the same groups.

5.8. LZWB1 vs. LZWB

From Table 5.13, we notice that LZWB1 has almost the same c.f. as lZWB. The c.t., d.t., and the counter output are smaller for LZWB1 than for LZWB. So, the theory behind LZWB1 worked but produced no overall higher c.f. than LZWB.

5.9. Conclusion

Based on the results of the previous sections, we conclude that LZW gives a higher c.f. than FAX for some groups and lower d.t. for all groups. So, an improvement in the LZW that increases the c.f. is desirable if LZW is to be used instead of FAX.

The techniques of compressing the run-lengths of the image instead of the image itself gave better c.f. and d.t. than those of FAX for g3 and g7. These techniques gave higher c.f. than LZW for g1 and g5. This means that more improvement in these techniques may produce a c.f. that is better than both LZW and FAX. Moreover, in the case that we are investigating which consists of black and white text and graphics, each pel is represented by 1 bit. So, it is envisioned that for the case of colored images where each pel is represented by more than one bit, the LZWBs methods will give better c.f. and they may be better than LZW and/or FAX.

6. MODIFICATIONS TO THE LZW ALGORITHM

Each entry in the LZW algorithm table consists of a string symbol and a character symbol that was previously encountered after this string. Reference [43] suggested using a table in which each entry consists of the symbols of two strings that were encountered after each other. This modification was chosen because it was expected that it would result in matching longer input strings to table entries. So, both LZW and the method suggested by [43] search for the longest string in the input that can be matched to a string encountered before; but the strings that are obtained by this method are predicted to be longer.

The search for the longest string in LZW is easy because after each successful match the string increases by one character. Hence, in LZW the search starts at symbol 256 and continues in one pass till all the table entries are searched. The search in this new method is not so easy because searching for the longest string requires the decomposition of every table entry that has as its first character the next unprocessed character in the input. Reference [43] did not show how it accomplished this task. In designing a code to do this task, the following two problems arose:

1) The first character of each table entry should be stored in a separate table so that only strings beginning with the required character are searched. Without this storing, it would be necessary to decompose each table entry just to see if it starts with the desired character or not; this results in a big increase in the compression time.

2) The decomposition of each table entry that begins with the desired character will take long searches; so, it

> is desirable to search for the longest block without the need to do these long searches.

In the following sections, we will propose two new methods that we will call LZW1 and LZW2 and that search for the longest string without decomposing every table entry that begins with the next unprocessed input character. Next, a method of decomposing every possible table entry will be presented. This method, that we will call LZW3, follows the concept suggested in [43]; nevertheless, it is not clear if [43] designed the details of the method in the same way we did. Actually, [43] never showed how to get the longest string, although this is a critical point in applying the concept that [43] proposed.

The following definitions are used in the following discussions and in the code used to simulate the above three methods:

L_i = The last string sent to the output.

L_j = The current longest string to be sent to the output.

w_1 = The first symbol of a table entry.

w_2 = The second symbol of a table entry.

w_3 = The first character of w_2 in a table entry.

first_char = The first character in w_2 while searching for the longest block.

code(w1, w2) = The code of the tables index corresponding to "w_1,w_2". It is found by a scan function.

The variables w_3 and first_char are used to solve the first of the two problems mentioned above. Since these two variables represent a character, 8 bits are needed to address each of them. The variable w_2 represents a string symbol; hence, at least 12 bits are needed to address w_2 in the case of a 4096 entry table. To simulate a table where each entry consists of w_1, w_2, and w_3, three tables were used. Two of these tables, where in these two each entry is an unsigned number, represent w_1 and w_2 and the third table is a table of characters that represent w_3.

Note that LZW used one table of unsigned numbers to represent w and a character table to represent k. The three tables mentioned above need more memory than the two tables of LZW. This explains the need to use the far pointers in coding these new methods. To make the code of LZW as close as possible to the code of its modifications, far pointers were also used in coding LZW although there was no need for these far pointers.

6.1. Method LZW1

Method LZW1 avoids using long searches, used in LZW3 later, by firstly, finding the longest string it can build character by character, i.e., it will search the w_2 table with w_2 only equal to one of the character symbols. Secondly, it enters a second loop where it searches for a string that begins with the current string and that matches the input. If it finds that string, this string will be the LZW1 current string, and this second loop will start again. If no string, that begins with the current string and matches the input, was found, the current string will be in this case the longest string we can get. Hence, it will be sent out, the tables will be updated, and LZW1 will start again in the first loop. The coding of LZW1 can be described as follows, in a C language like code:

```
1.  in_index = out_index = 0;
2.  $L_1$ = input[in_index++];
    output[out_index++] = $L_i$;
    $L_j$ = input[input_index++];
    first_char = $w_1$ = $w_3$ = $L_j$;
3.  while (in_index < bufr_size)
          {
          while (in_index < bufr_size)
            {
            $w_2$ = input[input_index++];
            if (string "wl,w2" is in the tables]
                $w_1$ = code(wl,w2);
            else
                first_char = $w_2$;
            }
          while (in_index < bufr_size)
            {
            start from "position" and search wl_table and
            w3_table for symbol "code" that corresponds
            to wl and first_char0j;
            if(tables has $w_1$ as first string and second string
```

```
            starts with first_char, i.e., corresponding $w_3$ =
            first_char)
                {
                position = code + 1;
                find $w_2$ at the matched symbol code;
                decompose $w_2$ into characters;
                if ($w_2$ matches the input)
                    {
                    $w_1$ = code;
                    adjust in_index;
                    first_char = input[input_index++];
                    }
                }
           else
                break
           }
      $L_j$ = $w_1$;
      output[out_index++] = $L_j$;
      update tables w1_table, w2_table, and w3_table with $L_i$,
      $L_j$, and $w_3$, respectively;
      $L_i$ = $L_j$;
      $w_1$ = $w_3$ = first_char;
      }
4.    END.
```

The decompression is straightforward and can be described as follows:

```
1.  in_index = out_index = 0;
2.  while (in_index < input_size)
            {
            $w_2$ = input[in_index++];
            decompose($w_2$);
            update decompress buffer with characters from $w_2$
            decomposition;
            update w1_table and w2_table with $w_1$ and $w_2$;
            $w_1$ = $w_2$}
3.  END.
```

Appendix H contains the listing of the LZW1 code.

6.2. Method LZW2

Method LZW2 does more searching than LZW1 in order to get the longest string. It also consists of a "while" loop that contains two smaller "while" loops. The outer and first "while" loops are similar to the ones in LZW1. The second "while" loop is different.

In LZW1, the second "while" loop can be summarized as follows:

```
while (more input and more table entries are to be searched)
      {
      read next character element in the input string;
      match the input string to a table entry that has $w_1$
      as its first string and first_char as first character
```

```
        of the entry second string;
        let $w_1$ = symbol of the matched entry;
        }
```

In LZW2, the second "while" loop can be summarized as follows:

```
    while (more input and more table entries are to be searched)
        {
        read next character element in the input string;
        loop till you find the longest string that matches
        the input and has $w_1$ as its first string and first_char
        as the first character of its second string;
        let $w_1$ = symbol of the longest matched string;
        }
```

The decompression of LZW2 is exactly the same as of LZW1. Appendix I contains the listing of the LZW2 code.

6.3. Method LZW3

Method LZW3 searches in the LZW table for the longest possible string. It searches every single element that has w_1 as its first string and its second string w_2 starts with first_char. To make the search more efficient, we also make a table for the second character of w_2 and use this information to speed up the search. In the results, we will see that even with this improvement, LZW3 takes a very long time without producing a considerable increase in the compression factor. The decompression process of LZW3 is exactly the same as of LZW1. Appendix J contains the listing of the LZW3 code.

6.4. Results of Compression Using LZW1, LZW2, and LZW3

To compare the LZW, LZW1, LZW2, and LZW3 methods, we apply them to an image that has an infinite size and consists of a repetition of the same byte, e.g., black or white images. From manually tracing the methods, we observe that after sending n symbols from each method, these symbols represent a total number of input bytes, we will call "sum", as follows:

1) LZW: $\text{sum} = 1+2+3+4+\dots\dots\dots\dots+n$

which can be expressed as

$$\text{sum} = \frac{n(n+1)}{2}$$

2) LZW1: $\text{sum} = 1+1+2+2+4+4+8+8+16+16\dots\dots\dots\dots+2^{\text{int}((n-1)/2)}$

which can be expressed for $n = 2m$ as

$$\begin{aligned}\text{sum} &= 2(1+2+4+8+\dots\dots\dots\dots+2^{m-1})\\ &= 2\,\frac{2^m-1}{2-1}\\ &= 2^{m+1} - 2\\ &= 2^{(n/2)+1} - 2 \quad ; \quad n = 2, 4, 6, 8 \dots\end{aligned}$$

3) LZW2: $\text{sum} = 1+1 +2+2 +4 +6+6 +12 +18+18 +36 +54+54 +108\dots$

for $n = 2 + 3m$ and $n >= 5$ we get

$$\begin{aligned}\text{sum} &= 1+1 +(2+2+6+6+18+18+54+54+\dots)\\ &\qquad +(4+12+36+108+\dots)\\ &= 1+1 +4(1+3+9+27+\dots) +4(1+3+9+27+\dots)\end{aligned}$$

$$= 2 + 8(1+3+9+27+....+3^{m-1})$$

$$= 2 + 8(1+3+9+27+....+3^{(((n-2)/3)-1)})$$

;n=5,8,11,14,...

$$= 2 + 8 \frac{3^{(\frac{n-2}{3})} - 1}{3-1}$$

$$= 2 + 4(3^{(\frac{n-2}{3})} - 1)$$

;n=5,8,11,14,...

4) LZW3: sum = $1+1+2+3+5+8+13+21..........+a_n$

from [45], we get

$$a_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right)$$

where $a_0 = 0$, $a_1 = 1$, $a_2 = 1$, $a_3 = 2$, and so on.

These terms can be summed as two geometrical series. Hence, after rearranging, we get:

$$\text{sum} = \frac{2}{\sqrt{5}} \left\{ \frac{(\frac{1+\sqrt{5}}{2})^n - 1}{\sqrt{5} - 1} + \frac{(\frac{1-\sqrt{5}}{2})^n - 1}{\sqrt{5} + 1} \right\}$$

Table 6.1 contains the results of sum with respect to some values of n for LZW, LZW1-LZW3. These values are drawn in Figures 6.1 and 6.2. From the above table and figures, we see that for small values of n, LZW gives higher value of sum than the other methods. LZW3 crosses LZW at almost n = 6 and then rises very fast. LZW1 and LZW2 cross LZW at almost n = 9 and 8, respectively, then rise but not as fast as LZW3, with LZW2 being the highest. We will use these results in our analysis of the

Table 6.1. Size of the data represented by n symbols for each LZWx method

n	LZW	LZW1	LZW2	LZW3
1	1	1	1	1
2	3	2	2	2
3	6	4	4	4
4	10	6	6	7
5	15	10	10	12
6	21	14	16	20
7	28	22	22	33
8	36	30	34	54
9	45	46	52	88
10	55	62	70	143
11	66	94	106	232
12	78	126	160	376
13	91	190	214	609
14	105	254	322	986
15	120	382	484	1596
16	136	510	646	2583
17	153	766	970	4180
18	171	1022	1456	6764
19	190	1534	1942	10945
20	210	2046	2914	17710
21	231	3070	4372	28656
22	253	4094	5830	46367
23	276	6142	8746	75024
24	300	8190	13120	121392
25	325	12286	17494	196417
26	351	16382	26242	317810
27	378	24574	39364	514228
28	406	32766	52486	832039
29	435	49150	78730	1346268
30	465	65534	118096	2178308
31	496	98302	157462	3524577
32	528	131070	236194	5702886
33	561	196606	354292	9227464
34	595	262142	472390	14930351
35	630	393214	708586	24157816
36	666	524286	1062880	39088168
37	703	786430	1417174	63245985
38	741	1048574	2125762	1.0E+08
39	780	1572862	3188644	1.7E+08
40	820	2097150	4251526	2.7E+08

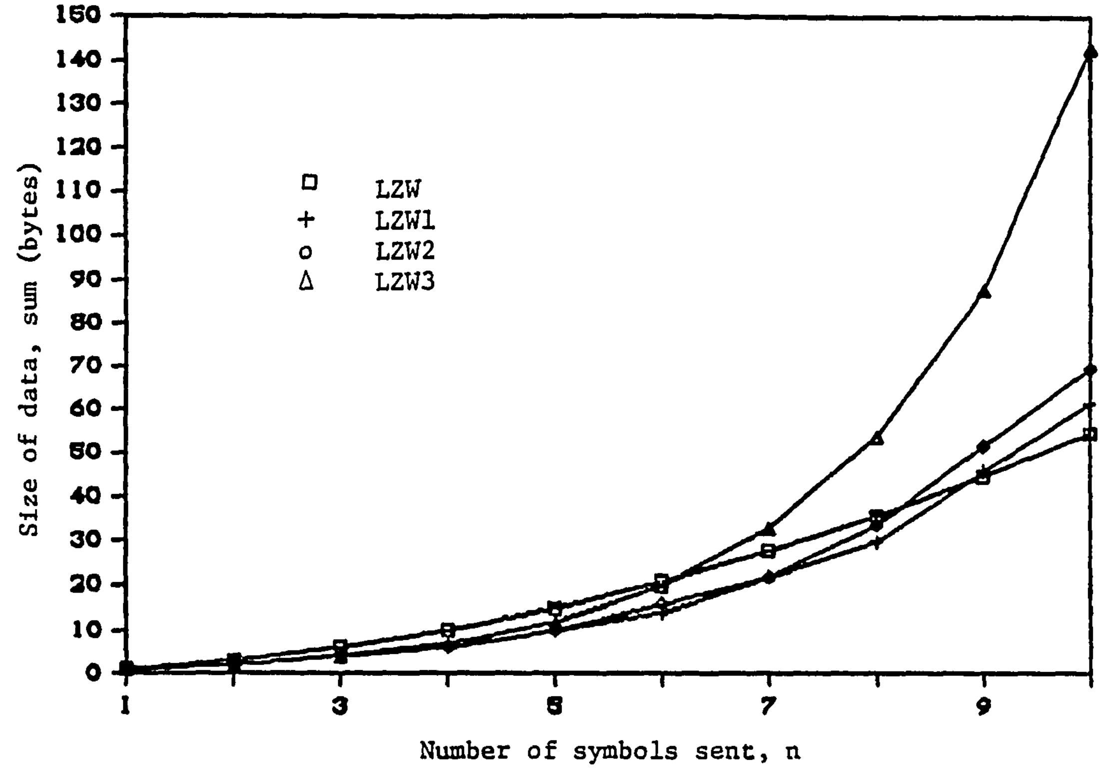


Figure 6.1. Plot of size of data (sum) vs. number of symbols used (n) for compressing a white image of infinite size (n = 1 to 10)

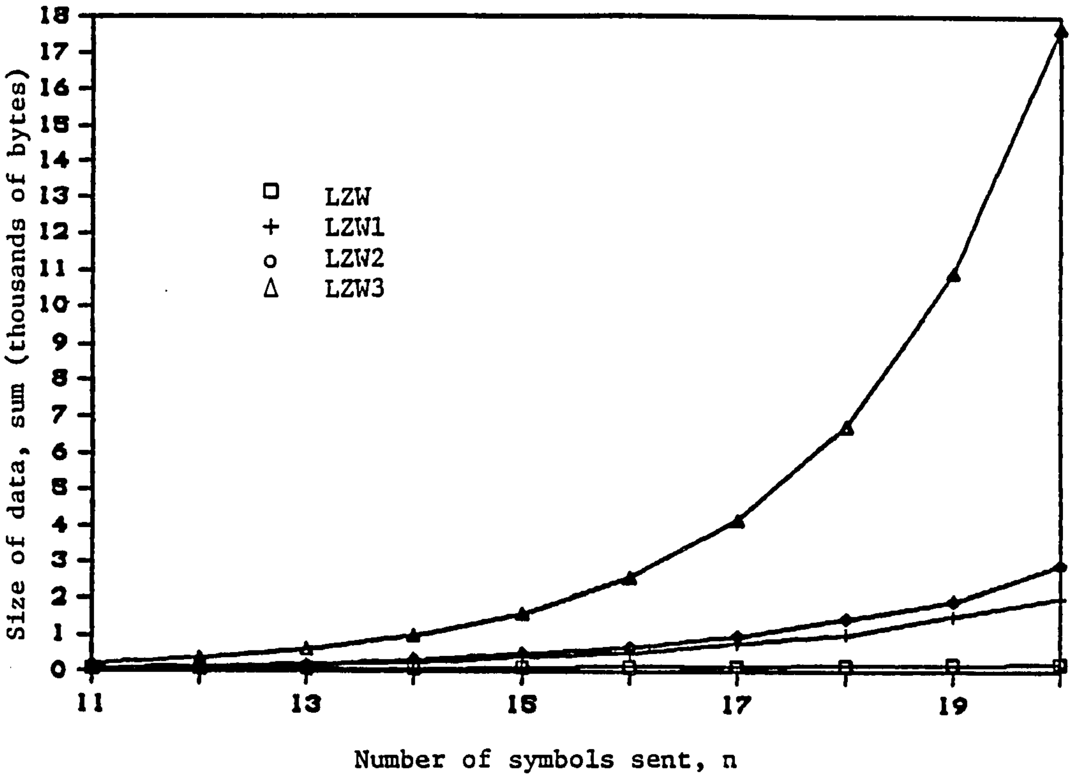


Figure 6.2. Plot of size of data (sum) vs. number of symbols used (n) for compressing a white image of infinite size (n = 11 to 20)

images compression results. Note that although this theoretical treatment shows a big difference between the methods for the infinite image, the results are not the same for an image of limited size. Table 6.2 contains the results of compressing a white screen using each of the LZW methods. This table shows that there is no big difference in the c.f. of the 4 LZW methods.

The results of compressing each group by methods LZW1, LZW2, and LZW3 are presented in Tables 6.3, 6.4, and 6.5, respectively. Table 6.6 contains the results of compressing each group using "LZW3+LZWB1" which is similar to LZWb1 but with the LZW3 used instead of LZW. From these tables and the corresponding tables for LZW and FAX, we get the following remarks:

1) The results of LZW2 and LZW1 are very close to LZW. LZW1 and LZW2 have very small advantage in c.f. and table size. LZW2 has slightly higher c.t. than LZW1. The c.t. of both methods are slightly higher than the c.t. of LZW. The table size of both LZW1 and LZW2 are very slightly higher than LZW. Taking all the groups into consideration, it seems that LZW1 and LZW2 give better c.f. and d.t. than lZW.

2) LZW3 gives better c.f. than LZW for all groups except g3. The d.t. of LZW3 is similar to LZW but its c.t. is very big. In fact, the c.t. of LZW3 is bigger than one minute; for this reason, we do not include c.t. in the tables of LZW3.

3) LZW gives better c.f. than LZW1 and LZW2 for g3 and g4. This can be explained by using the theoretical analysis we presented before.

Table 6.2. Results of compressing a white screen using methods FAX, LZW, LZWL, LZW2, and LZW3

Method	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
FAX	305.49	2.79	1.54	NA[a]	NA
LZW	59.48	8.95	1.59	434	0
LZW1	340.43	2.64	1.48	286	0
LZW2	363.64	3.13	1.49	284	0
LZW3	410.26	5.44	1.54	281	0

[a] NA = entry not valid for this method.

Table 6.3. Results of compressing each group of the image data using method LZW1

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	6.29	0.64	1.10	13.32	1.20	2118	49
GROUP 2	5.72	0.68	1.00	13.43	1.37	2209	0
GROUP 3	3.60	1.28	0.95	29.14	1.46	3388	268
GROUP 4	3.62	0.91	0.98	22.60	1.36	3127	33
GROUP 5	5.31	0.55	1.10	3.20	0.47	941	0
GROUP 7	4.27	1.15	1.02	30.28	1.46	3208	308
AVERAGE	4.80	0.87	1.03	18.66	1.22	2499	110

Table 6.4. Results of compressing each group of the image data base using method LZW2

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	6.35	0.64	1.11	15.04	1.20	2113	48
GROUP 2	5.78	0.69	1.01	15.16	1.36	2187	0
GROUP 3	3.62	1.29	0.96	32.82	1.47	3386	260
GROUP 4	3.64	0.92	0.99	24.25	1.35	3113	31
GROUP 5	5.40	0.56	1.12	3.68	0.43	936	0
GROUP 6	4.30	1.16	1.03	29.20	1.46	3200	309
AVERAGE	4.85	0.88	1.04	20.03	1.21	2489	108

Table 6.5. Results of compressing each group of the image data base using method LZW3

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Dcmprs. time s	Table size	Extra calls
GROUP 1	6.62	0.67	1.16	1.43	2056	39
GROUP 2	6.12	0.73	1.07	1.57	2048	0
GROUP 3	3.74	1.33	0.99	1.59	3345	216
GROUP 4	3.77	0.95	1.02	1.48	3047	14
GROUP 5	5.82	0.61	1.21	0.47	894	0
GROUP 7	4.54	1.23	1.08	1.59	3149	268
AVERAGE	5.10	0.92	1.09	1.36	2423	90

Table 6.6. Results of compressing each group of the image data base using method LZW3 combined with LZWB1

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Dcmprs. time s	Table size	Extra calls
GROUP 1	6.06	0.61	1.06	1.01	2215	182
GROUP 2	5.53	0.66	0.97	0.98	2323	0
GROUP 3	3.00	1.07	0.79	2.04	3567	858
GROUP 4	3.34	0.84	0.91	1.28	3294	69
GROUP 5	5.77	0.60	1.20	0.27	930	0
GROUP 7	4.14	1.12	0.99	1.74	3261	451
AVERAGE	4.64	0.82	0.98	1.22	2598	260

Since the analysis showed that LZW is better than LZW1 and LZW2 for small values of n adding to that the fact that g3 and g4 contain a lot of text (which means the run-lengths of these two groups consist of small runs), the length of the strings LZW1 and LZW2 produce in the LZW table is small, hence LZW is better.

4) The same conclusion reached in 3 about LZW1 and LZW2 can be reached for LZW3. But as the calculation shows, LZW3 crosses with LZW for smaller values of n and does much better than LZW for bigger values of n; hence, in general, LZW3 is better than lZW. Table 6.5 showed that LZW3 always had bigger c.f. than LZW except for g3 where the c.f. of the two methods were very close to each other.

4) The c.f. of LZW1-LZW3 compared to FAX are, as was the case for LZW, higher for g3 and g5 and lower for the other groups. The ratio of the c.f. of LZW3 to that of FAX is 1.35 for g3 which is screens full of text. This big gain in c.f. for g3 justifies using LZW3 at least for g3.

5) From Table 6.6, it is clear that the only advantage LZW3+LZWB1 has over LZW3 is a slightly less d.t. LZW3+LZWB1 has the disadvantage of lower c.f. and slightly bigger table size. Compared to LZWB1 alone, LZW3_LZWB1 gives a higher c.f. The same analysis and conclusion we got for LZWB1 in Chapter 4 applies to LZW3+LZWB1.

7. METHODS R8, R4, AND BIG

7.1. Method R8

The following observations led to the development of methods R8 and R4:

1) LZW gives higher c.f. if the input contains repeated strings and strings that can be built from each other. The methods LZWBs were an attempt to change the input data to LZW from just the pels of the screen in their regular form to other form, run-lengths symbols, that might result in a higher c.f. using LZW. As was shown in Chapter 5, this attempt was successful for some groups and not successful for others. So, another attempt to produce better input to LZW was developed by the author.

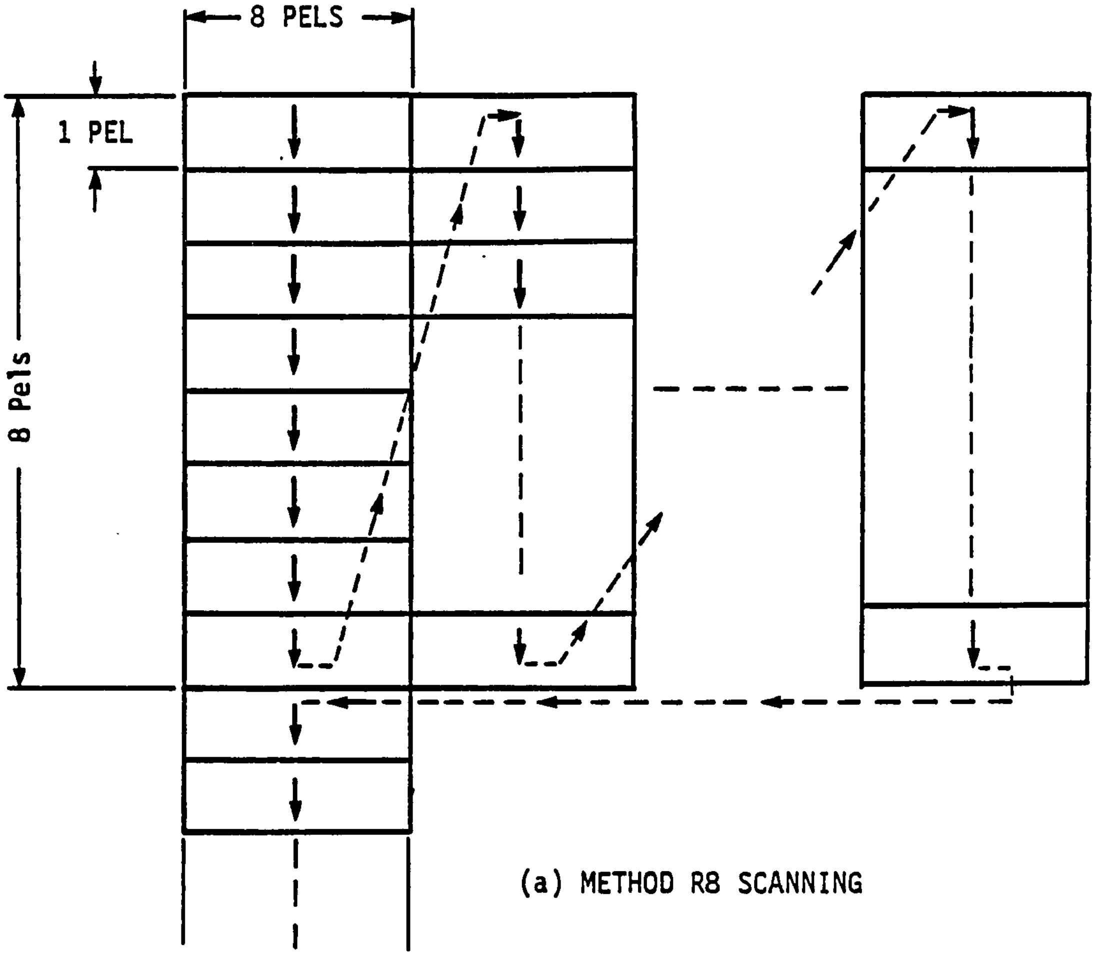
2) The attempt of Chapter 6 to produce better versions of LZW gave modified versions of LZW (namely, LZW1, LZW2, and LZW3) that gave better c.f. than LZW but not as high as expected.

3) LZW, LZW1, LZW2, and LZW3 gave better c.f. than FAX for g3 which consists of screens full of text. At the first glance, it seems that groups consisting of mainly graphical data, and not g3, should give higher c.f. because there is no relation between the screen bytes in the case of g3. But, besides the fact that FAX is not optimum for screens that have a lot of small white and black runs, a closer look at the functioning of LZW and the structure of the input data suggests that LZW does better than FAX for g3 because LZW benefits from the dependency

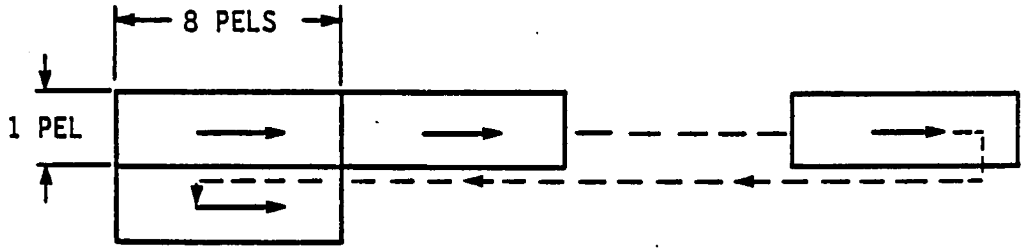
between the characters themselves. That is to say, if character "B" comes after "A", the rows of pels representing "B" come after the rows of pels representing "A". This results in adding, to the LZW table, a number of strings equal to the character height (assume from now on that the character height is 8). So, the next time "B" comes after "A", LZW will detect that 8 strings have already been encountered before and are in the table. Hence, LZW represents these 8 strings with fewer symbols than in the case of an input from the normal scan. Note that at this point LZW denotes any of LZW, LZW1, LZW2, and LZW3.

Taking the above 3 points into consideration, we developed methods R8 and R4. Method R8 can be explained as follows.

Instead of reading the screen in the normal scan, R8 divides the screen into blocks of 8 lines and reads each block column by column, where a column width is one byte. Figure 7.1 represents the normal scan and the scan in method R8. So, method R8 is not a compression method; it is only a way of arranging the screen data in the best form for compression. Consequently, method R8 (similarly, R4) should be used with any LZW method. The notation for using LZW combined with R8 will be "LZW+R8". Throughout the rest of the thesis the notation LZWx will be used to denote LZW, LZW1, LZW2, or LZW3 (so, x = 0, 1, 2, or 3 with LZW0 denoting LZW). The notation Ry will be used to denote R8 or R4. The letter "R" in the method name stands for "rotated" scan. The numbers 4 and 8 stand for the column width in pels.



(a) METHOD R8 SCANNING



(b) NORMAL SCANNING

Figure 7.1. A comparison between normal scanning and scanning of method R8

7.2. Method R4

Method R8 was designed with the screen viewed as characters in order to increase the c.f. of compressing textual data. But for graphics screens or blocks this view may not be the best idea for compression. To investigate this point, we developed method R4. R4 works similar to R8 except that the column width in the rotated scan is 4 pels or half a byte. It is envisioned that this will work better for graphics data because it can isolate longer strings, specially runs of black pels.

Another reason for developing R4 is that such a scanning method might be necessary when scanning typed material where the character width of each letter is not the same for all letters.

7.3. Method BIG

LZW is known to work better as the input data size increases, up to a certain limit [41]. In all the previous LZWx methods, we compress a screen or part of a screen; this means that the input data maximum size is 16 KB. The previous methods (e.g., LZWx+Ry) results showed that the table size was smaller than the table maximum size. This means, as will be cleared later, there is a room for increasing the input size. In method BIG, we use any of the previous methods to compress more than one screen. So, BIG is not an actual method but we name it as a method to make the comparison and investigation clearer.

7.4. Results and Analysis of R8 and R4

Tables 7.1 and 7.2 contain the results of using LZW with R8 and R4, respectively. Tables 7.3 and 7.4 contain the results of using LZW1 with R8 and R4, respectively. Tables 7.5 and 7.6 contain the results of using LZW2 with R8 and R4, respectively. Tables 7.7 and 7.8 contain the results of using LZW3 with R8 and R4, respectively.

From the above mentioned tables, we get the following points:

1) For all groups, Ry+LZW3 gives higher c.f. than Ry+LZWx (where x=0, 1, 2) and LZWx without Ry.

2) c.f. of R8 vs. c.f. of R4: the c.f. results of the different groups can be classified as follows:

    a) For g1, R4+LZW or R4+LZW1 is almost the same as R8+LZW or R8+LZW1, respectively, and R4+LZW2 or R4+LZW3 is better than the R8+LZW2 or R8+LZW3, respectively.

    b) For g2, R4 is better than R8 when any of them is combined with LZW1, LZW2, or LZW3. For the LZW, R8 is better.

    c) For g3, R8 is better than R4 for any LZWx.

    d) For g4, R4 is better than R8 for any LZWx.

    e) For g5, R4 is better than R8 for LZW1-LZW3 and R8 is better than R4 for LZW.

    f) For g7, R4 is better than R8 for LZW1 and LZW2, same as R8 for LZW3. For g7, using LZW, R8 is better than R4.

    From the above classification, it is clear that, as expected, R8 is better than R4 when the data is only, or mostly, a textual screen. But for graphical data, R4 is better. When the data are a combination of text and graphics R4 is better or at least the same as R8 for all the LZWx methods except LZW.

Table 7.1. Results of compressing each group of the image data base using method LZW combined with method R8

Group #	Comprs. factor	$\frac{\text{C.F.}}{\text{FAX}}$	$\frac{\text{C.F.}}{\text{LZW}}$	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	6.96	0.70	1.21	14.58	1.79	1706	0
GROUP 2	6.70	0.80	1.18	16.78	2.09	1928	0
GROUP 3	5.84	2.08	1.54	20.09	2.09	2313	0
GROUP 4	4.41	1.11	1.20	24.33	1.99	2761	0
GROUP 5	5.44	0.57	1.13	3.89	0.67	855	0
GROUP 6	4.67	1.26	1.11	24.01	2.07	2882	167
AVERAGE	5.67	1.09	1.23	17.28	1.78	2074	28

Table 7.2. Results of compressing each group of the image data base using method LZW combined with method R4

Group #	Comprs. factor	$\frac{\text{C.F.}}{\text{FAX}}$	$\frac{\text{C.F.}}{\text{LZW}}$	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	6.95	0.70	1.21	15.10	2.06	1726	0
GROUP 2	6.61	0.79	1.16	17.67	2.41	1959	0
GROUP 3	5.40	1.92	1.42	22.65	2.39	2513	24
GROUP 4	4.67	1.18	1.27	24.32	2.25	2620	0
GROUP 5	5.47	0.57	1.13	4.07	0.76	859	0
GROUP 7	4.59	1.24	1.10	25.74	2.38	3207	208
AVERAGE	5.62	1.07	1.22	18.26	2.04	2147	39

Table 7.3. Results of compressing each group of the image data base using method LZW1 combined with method R8

Group #	Comprs. factor	C.F. FAX	C.F. LZW	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	7.51	0.76	1.31	7.83	1.16	1598	0
GROUP 2	6.78	0.81	1.19	10.21	1.33	1895	0
GROUP 3	7.58	2.70	2.00	11.68	1.43	1974	0
GROUP 4	4.71	1.19	1.28	17.46	1.31	2698	0
GROUP 5	5.80	0.60	1.20	2.42	0.44	839	0
GROUP 7	4.86	1.31	1.16	25.10	1.43	2948	228
AVERAGE	6.21	1.23	1.36	12.45	1.18	1992	38

Table 7.4. Results of compressing each group of the image data base using method LZW1 combined with method R4

Group #	Comprs. factor	C.F. FAX	C.F. LZW	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	7.52	0.76	1.31	7.63	1.17	1590	0
GROUP 2	6.95	0.83	1.22	9.72	1.35	1864	0
GROUP 3	6.98	2.48	1.84	13.05	1.44	2127	21
GROUP 4	5.19	1.31	1.41	14.83	1.32	2453	0
GROUP 5	6.02	0.63	1.25	2.33	0.45	824	0
GROUP 6	4.94	1.34	1.18	24.51	1.45	2905	249
AVERAGE	6.27	1.22	1.37	12.01	1.20	1961	45

Table 7.5. Results of compressing each group of the image data base using method LZW2 combined with method R8

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	7.56	0.76	1.32	9.04	1.16	1589	0
GROUP 2	6.84	0.82	1.20	11.65	1.36	1884	0
GROUP 3	7.64	2.72	2.02	14.37	1.42	1970	0
GROUP 4	4.65	1.17	1.26	18.98	1.32	2701	0
GROUP 5	5.83	0.61	1.21	2.78	0.41	837	0
GROUP 7	4.82	1.30	1.15	21.80	1.44	2944	226
AVERAGE	6.22	1.23	1.36	13.10	1.19	1988	38

Table 7.6. Results of compressing each group of the image data base using method LZW2 combined with method R4

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Comprs. time s	Dcmprs. time s	Table size	Extra calls
GROUP 1	7.64	0.77	1.33	8.91	1.18	1521	0
GROUP 2	6.91	0.82	1.21	11.33	1.37	1877	0
GROUP 3	6.59	2.35	1.74	15.88	1.47	2133	15
GROUP 4	5.12	1.29	1.39	17.03	1.31	2474	0
GROUP 5	6.06	0.63	1.26	2.65	0.41	822	0
GROUP 7	4.99	1.35	1.19	21.92	1.45	2893	249
AVERAGE	6.22	1.20	1.35	12.95	1.20	1953	44

Table 7.7. Results of compressing each group of the image data base using method LZW3 combined with method R8

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Dcmprs. time s	Table size	Extra calls
GROUP 1	8.07	0.82	1.41	1.48	1515	0
GROUP 2	7.33	0.87	1.29	1.57	1781	0
GROUP 3	7.84	2.79	2.07	1.58	1915	0
GROUP 4	4.93	1.24	1.34	1.51	2599	0
GROUP 5	6.41	0.67	1.33	0.51	790	0
GROUP 7	5.30	1.43	1.26	1.63	2835	200
AVERAGE	6.65	1.30	1.45	1.38	1906	33

Table 7.8. Results of compressing each group of the image data base using method LZW3 combined with method R4

Group #	Comprs. factor	C.F. / FAX	C.F. / LZW	Dcmprs. time s	Table size	Extra calls
GROUP 1	8.17	0.83	1.43	2.14	1506	0
GROUP 2	7.41	0.88	1.30	2.40	1773	0
GROUP 3	7.31	2.60	1.93	2.41	2075	0
GROUP 4	5.41	1.36	1.47	2.28	2387	0
GROUP 5	6.60	0.69	1.37	0.72	783	0
GROUP 7	5.30	1.43	1.26	2.43	2827	228
AVERAGE	6.70	1.30	1.46	2.06	1892	38

Although the c.f. ratios (LZW3/FAX) and (LZW3/LZW) seem to be the same for R4 and R8 when combined with LZW3, the average c.f. of all groups is higher in the case of R4 (6.7 for R4 vs. 6.65 for R8).

3) R4 has higher d.t. than R8 when any of them is combined with LZW or LZW3 and almost the same as R8 when any of them is combined with LZW1 or LZW2. The d.t. of R8+LZW3 is approximately 2/3 of the d.t. of R4+LZW3.

   d.t. of R8+LZWx (x=1, 2, 3) are less than d.t. of R8+LZW with R8+LZW1 and R8+LZW2 having the smallest values.

   d.t. of R4+LZW1 or R4+LZW2 are less than d.t. of R4+LZW. d.t. of R4+LZW3 is the same as d.t. of R4+LZW.

   So, for Ry+LZWx (x=1, 2, 3), although LZW1-LZW3 have longer strings to be decomposed than LZW, the number of strings in the case of LZW1-LZW3 is less, resulting in a d.t. smaller than or equal to the d.t. of LZW.

4) Although unexpected, the c.t. of Ry+LZW1 or Ry+LZW2 are smaller than the c.t. of Ry+LXW. Most of the c.t. of LZW3 or Ry+LZW3 are longer than one minute, so it was decided not to include them in the tables.

5) The table size for Ry+LZWx decreases as x increases. The table size of R4+LZWx is close to the table size for R8+LZWx for each corresponding value of x.

6) For g3, the c.f. of R8+LZWx increases as x increases. R4 has a similar trend except for R4+LZW2, where the c.f. is less than R4+LZW1 but still higher than LZW.

7) The c.f. of Ry+LZW3 is higher than FAX for g3, g4, and g7 and less for easy graphics such as g2, g5, and g1 which is mixed of text and easy graphics. The result of compressing g1 can be explained by the fact that the majority of the documents in g1 are easy graphics; only document 4 can be considered as a "text only" document. Hence, the effect of documents totally or partially consisting of graphics cause the c.f. of FAX to be higher than Ry+LZW3.

   The highest ratio of the c.f. of Ry+LZWx to FAX c.f. is for R8+LZW3 where it is 2.79.

8) LZW1 or LZW2 when combined with Ry give c.f. that are smaller than LZW3+Ry by no more than 10%; but they have

the advantage of lower d.t. and extremely lower c.t. in comparison to LZW3. So, if the c.t. is not important, as in our case, LZW3+Ry is the best choice. Choosing between R4 and R8 depends on the group of data to be compressed and the d.t. allowed. But as we saw before, LZW3+R4 gives an overall c.f. that is higher than LZW3+R8 and its d.t. is only in the range of 2 s (=1.5 times the d.t. of LZW3+R8). Hence, we think LZW3+R4 should be the choice.

Furthermore, R8 may not do as well for variable width characters as it did in the case of g3 as shown in Tables 7.7.

If the c.t. is important, Ry+LZW1 or Ry+LZW2 is the choice. From the previous data and analysis, there is no big difference between Ry+LZW1 and Ry+LZW2, and choosing any of them will do as well as the other.

7.5. Results and Analysis of BIG

To investigate BIG, we grouped two or more files for a total of 19 groups or combinations. To avoid confusion with the group numbering that we made in Chapter 3, we call these "combinations" and denote them by c1, c2,... etc. Table 7.9 lists these combinations and the images they combine. The images in each combination are listed in their compression order. Tables 7.10-7.12 contain the c.f. results of BIG+Ry+LZWx (x=0, 2, and 3). Table 7.13 contains the c.t. results of BIG+Ry+LZW and BIG+Ry+LZW2. Since the c.t. results of BIG+Ry+LZW3 are bigger than 1 min, they will not be included. Table 7.14 contains the summation of the c.t. of the individual images in each combination when each individual image is compressed alone using Ry+LZW and Ry+LZW2. Table 7.15 contains the extra calls made when compressing each combination. The presence of negative values of the "extracalls" is used to denote that

Table 7.9. The combinations used in BIG

Combination #	Image 1	Image 2	Image 3	Image 4	Image 5
1	doc1a	doc1b	doc1c		
2	doc2a	doc2c	doc2c		
3	doc4a	doc4b			
4	doc6a	doc6b			
5	doc1a	doc1b			
6	doc2a	doc2b			
7	doc4a	doc4b	romtxt		
8	doc4a	doc4b	frnch2a		
9	doc4a	doc4b	doc4c	romtxt	frnch2a
10	doc4a	doc4b	doc4a		
11	doc4a	doc4b	doc2a		
12	doc4a	doc4b	cprog		
13	doc4a	doc2a	doc4b		
14	doc4a	electrc	doc4b		
15	doc6a	doc6b	doc8		
16	doc6a	doc6b	frnch3a		
17	doc6a	doc6b	electrc		
18	doc6a	doc6b	flowchrt		
19	doc6a	doc6b	flowchrt	electrc	

Table 7.10. Compression factor results using Ry+LZW

Combination	R8+LZW				R4+LZW			
#	BIG	IND	BIG/IND	BIG/FAX	BIG	IND	BIG/IND	BIG/FAX
1	6.70	6.07	1.10	1.00	6.61	6.05	1.09	0.99
2	11.49	9.34	1.23	0.77	11.37	9.35	1.22	0.76
3	5.97	5.09	1.17	3.51	5.54	4.72	1.17	3.26
4	8.49	7.55	1.12	0.71	8.65	7.82	1.11	0.72
5	5.09	4.80	1.06	1.01	5.00	4.77	1.05	1.00
6	9.80	8.61	1.14	0.73	9.79	8.66	1.13	0.73
7	4.34	4.85	0.89	2.73	2.54	4.53	0.56	1.60
8	4.07	4.45	0.91	2.29	2.84	3.74	0.76	1.60
9	2.78	4.44	0.63	1.67	2.57	3.84	0.67	1.55
10	6.61	5.16	1.28	3.78	6.16	4.79	1.29	3.52
11	6.70	5.98	1.12	2.78	6.11	5.63	1.09	2.54
12	6.68	6.09	1.10	3.01	5.25	5.64	0.93	2.36
13	6.78	5.98	1.13	2.81	6.51	5.63	1.16	2.70
14	5.23	4.91	1.07	2.91	4.94	4.59	1.08	2.74
15	8.46	7.54	1.12	0.59	8.62	7.77	1.11	0.60
16	8.89	7.52	1.18	0.74	8.85	7.59	1.17	0.74
17	6.65	6.21	1.07	1.46	6.55	6.19	1.06	1.43
18	7.48	6.66	1.12	0.86	7.51	6.81	1.10	0.87
19	5.54	5.99	0.92	1.16	5.07	5.98	0.85	1.06

Table 7.11. Compression factor results using Ry+LZW2

Combination #	R8+LZW2				R4+LZW2			
	BIG	IND	BIG/IND	BIG/FAX	BIG	IND	BIG/IND	BIG/FAX
1	6.68	6.14	1.09	1.00	6.73	6.30	1.07	1.01
2	10.97	9.05	1.21	0.74	11.10	9.18	1.21	0.75
3	8.67	7.04	1.23	5.10	8.15	6.55	1.24	4.79
4	8.50	7.91	1.07	0.71	8.96	8.18	1.10	0.75
5	5.05	4.80	1.05	1.01	5.10	4.92	1.04	1.02
6	9.33	8.19	1.14	0.69	9.46	8.35	1.13	0.70
7	7.86	6.83	1.15	4.94	7.17	6.41	1.12	4.51
8	5.57	5.20	1.07	3.13	4.09	4.43	0.92	2.30
9	4.61	5.41	0.85	2.78	4.05	4.79	0.85	2.44
10	10.49	7.13	1.47	5.99	9.99	6.59	1.52	5.72
11	8.82	7.53	1.17	3.66	8.54	7.20	1.19	3.54
12	10.03	8.42	1.19	4.52	9.41	7.74	1.22	4.24
13	8.68	7.53	1.15	3.60	8.37	7.20	1.16	3.47
14	6.55	6.04	1.08	3.64	6.14	5.72	1.07	3.41
15	8.38	7.84	1.07	0.59	9.00	8.21	1.10	0.63
16	8.58	7.67	1.12	0.71	9.03	7.80	1.16	0.75
17	6.74	6.44	1.05	1.47	6.75	6.47	1.04	1.48
18	7.34	6.88	1.07	0.85	7.81	7.09	1.10	0.90
19	5.25	6.15	0.85	1.10	4.97	6.23	0.80	1.04

Table 7.12. Compression factor results using Ry+LZW3

Combination #	R4+LZW3				R4+LZW3			
	BIG	IND	BIG/IND	BIG/FAX	BIG	IND	BIG/IND	BIG/FAX
1	6.93	6.46	1.07	1.04	6.98	6.54	1.06	1.05
2	11.63	9.77	1.19	0.78	11.73	9.89	1.19	0.79
3	8.63	7.14	1.21	5.08	8.38	6.66	1.26	4.93
4	9.60	8.67	1.11	0.80	9.79	9.02	1.09	0.82
5	5.25	5.02	1.05	1.05	5.26	5.09	1.03	1.05
6	9.86	8.88	1.11	0.73	9.99	9.03	1.11	0.74
7	7.87	6.87	1.15	4.95	7.51	6.57	1.14	4.72
8	5.69	5.36	1.06	3.20	4.36	4.55	0.96	2.45
9	4.43	5.51	0.80	2.67	3.99	4.93	0.81	2.40
10	10.54	7.19	1.47	6.02	10.24	6.72	1.52	5.85
11	9.02	7.78	1.16	3.74	8.89	7.42	1.20	3.69
12	10.15	8.50	1.19	4.57	9.75	7.97	1.22	4.39
13	9.01	7.78	1.16	3.74	8.86	7.42	1.19	3.68
14	6.87	6.24	1.10	3.82	6.69	5.95	1.12	3.72
15	9.33	8.57	1.09	0.65	9.74	8.96	1.09	0.68
16	9.59	8.37	1.15	0.80	9.70	8.64	1.12	0.81
17	7.40	6.96	1.06	1.62	7.28	7.05	1.03	1.59
18	8.30	7.43	1.12	0.96	8.51	7.74	1.10	0.98
19	5.72	6.62	0.86	1.20	5.63	6.77	0.83	1.18

Table 7.13. Compression time of each combination using Ry+LZW and Ry+LZW2

Combination #	R8+LZW s	R4+LZW s	R8+LZW2 s	R4+LZW2 s
1	99	100	60	60
2	60	61	36	38
3	55	61	34	35
4	37	39	24	23
5	64	65	47	47
6	31	32	23	24
7	129	131	66	70
8	121	125	96	121
9	264	276	183	187
10	96	104	52	53
11	111	108	55	57
12	111	112	47	49
13	105	112	59	62
14	105	114	83	89
15	81	85	52	49
16	70	85	48	47
17	82	88	66	66
18	75	78	58	56
19	116	124	122	132

Table 7.14. Summation of the compression times of the images in each combination using Ry+LZW and Ry+LZW2

Combination #	R8+LZW s	R4+LZW s	R8+LZW2 s	R4+LZW2 s
1	58	60	36	36
2	41	42	23	24
3	44	48	26	27
4	30	31	18	17
5	46	46	32	31
6	26	27	17	17
7	66	72	40	42
8	69	83	56	67
9	98	115	75	87
10	66	72	38	40
11	55	60	34	36
12	60	66	31	33
13	55	60	34	36
14	64	70	44	46
15	45	46	28	26
16	45	46	29	27
17	50	53	37	36
18	48	50	33	30
19	68	72	51	49

Table 7.15. Extra calls required when compressing each combination using Ry+LZW, Ry+LZW2, and Ry+LZW3

Combination #	R8+LZW	R4+LZW	R8+LZW2	R4+LZW2	R8+LZW3	R4+LZW3
1	936	1001	948	907	775	739
2	-1059	-1028	-930	-963	-1093	-1115
3	-268	12	-1382	-1225	-1371	-1299
4	-1331	-1376	-1334	-1464	-1622	-1665
5	346	422	381	338	219	214
6	-1666	2434	-1557	-1589	-1679	-1707
7	3523	8738	227	615	222	417
8	4016	7422	1902	3981	1781	3496
9	15320	16904	7705	9290	8179	9518
10	995	1354	-794	-641	-809	-722
11	936	1393	-218	-98	-299	-247
12	947	2256	-655	-444	-692	-564
13	874	1072	-159	-24	-293	-238
14	2273	2633	1041	1365	816	936
15	-61	-132	-28	-290	-413	-560
16	-244	-229	-115	-300	-507	-547
17	966	1038	906	898	480	547
18	435	419	513	251	13	-88
19	3861	4567	4281	4745	3612	3732

there were no extra calls and the number given is equal to the table size minus the table maximum size, i.e., minus 4096.

Checking the c.f. results in Tables 7.10-7.12, we observe that the method BIG, in general, produced the desired increase in the c.f. We also observe that the trends in the three tables are very similar. Hence, we chose to comment on only Table 7.12 which has the results of using BIG+Ry+LZW3.

From Table 7.12, we see that the difference between using R4 and R8 is very small, except for c8. In general, as expected and explained before, R8 produces a higher c.f. for textual data and R4 produces a higher c.f. for graphics data. In the following, we will look at the results of BIG+R8+LZW3. We will refer to the results of BIG+R4+LZW3 when necessary.

1) Combinations c1, c2, c3, and c4 represent the combination of the parts of each of the CCITT documents. This means the images combined in each combination are related together. For this reason, the result of c1, c2, c3, and c4 shows an increase in the ratio of the c.f. if the combination is compressed at once, over the total c.f. if each image was compressed alone. Tables 7.10-7.12 denote this ratio by BIG/IND, and we will use this notation in the rest of the thesis. Among the BIG/IND ratios of c1, c2, c3, and c4, the highest ratio was that of c3. This is expected since this combination is a combination of two textual screens. Note that the c.f. of c3 is 8.63 which is higher than the c.f. if each screen was sent as an ASCII text. If each screen was sent as ASCII text, then the c.f. is given by

   $$c.f. = 16000/(80x25) = 8.0$$

   We should note that the two textual screens in c3 have only 24 lines each with the last line being blank characters. So, for a completely filled screen the c.f. may be a little less, or may be higher.

The fact that we get a c.f. of c3 that is higher than the c.f. if we send the screen as ASCII is a very interesting and important result. It means that without any pattern recognition we get a c.f. higher than the c.f. if pattern recognition is used.

2) Combinations c5 and c6 are each the combination of the first two images in c1 and c2, respectively. Their c.f. result shows that for such images, compressing three images is better than compressing two images in one combination.

3) Combinations c7 and c8 consist of doc4a and doc4b each, followed by romtxt and frnch2a, respectively. The BIG/IND ratio of c7 is higher than that of c8. This difference can be explained by the following remarks:

   a) The characters in frnch2a are different from the characters in doc4a and doc4b while the characters of romtxt are the same.

   b) The image frnch2a is a screen filled with 22 lines while the images doc4a, doc4b, and romtxt are textual pages with 24 lines as a text and line 25 is blank. This means that, first image romtxt is more similar to doc4a and doc4b than image frnch2a. Second, the compression of frnch2a will not be as good as any of the other images because it is not in the best form for R8, i.e., it does not consist of lines that are next adjacent to each other and frnch2a has characters of 8 pels high.

4) The ratio of the c.f. of c8 using BIG+R8+LZW3 to the c.f. of BIG+R4+LZW3 is the highest ratio in Table 7.12 for any combination.

5) Each of c9 and c19 represent a combination of 5 images of textual and graphics screens, respectively. In the result of both combinations, BIG/IND is less than 1 but BIG/FAX is bigger than 1. The fact that BIG/IND is less than 1 suggests that, as expected, the LZWx methods lose their adaptation if the input size increases beyond a certain limit.

6) Combination c10 shows how LZWx benefits from repeated strings and how it is highly adaptable. These two observations come from the fact that doc4a is the first and third image in this combination.

7) Combinations cl1 and cl2 have images doc4a and doc4b as their first two images and doc2a and cprog as their third image, respectively. Although the third image is a graphics image in cl1 and a textual image in cl2, both combinations have BIG/IND around 1.15. This also shows the adaptability of LZWx.

8) The second and third images in cl3 are the third and second images in cl1. It is interesting that with this flipping of the images order, the resulted c.f. are still almost the same. BIG+R4+LZW3 gives similar results.

9) Combinations cl3 and cl4 both have doc4a and doc4b as their first and third images, and their second image is a graphical screen. Both combinations give BIG/IND bigger than 1.10. This also shows the adaptability of LZWx.

10) In the combination cl5, the third image is completely different from the first two images and still BIG/IND is bigger than 1. This also shows the adaptability of LZWx.

11) Combinations cl6, cl7, and cl8 start each with two related graphics screens, namely, doc6a and doc6b, followed by a third image that is also a graphics screen. The BIG/IND is bigger than 1 in the three combinations. The BIG/IND ratio increases with the c.f. of the third image.

12) In most combinations, there were some extra calls made but this did not affect the c.f. very much.

13) The compression time of the document increases as its order in compression increases. The compression time for images other than the first image is usually longer than when compressing this image alone. This is due to the fact that the method takes longer time to search the table as the table size increases.

8. GENERAL ANALYSES

In the previous chapters, we looked at the methods when we developed them. In this chapter, we will present some general remarks about these methods.

8.1. Building the Screen

In Chapter 3, we defined group 6 as a group that contains an image that is built gradually and can be divided into smaller blocks. We saw in Chapter 4 that, when using FAX, this division does not increase the total c.f. of the small blocks. We did not look at this point for the methods LZWx in the last chapters. Table 8.1 presents the results of dividing the image pdraw3 into 4 smaller blocks using all the methods developed so far.

From Table 8.1, we conclude that LZWx does not benefit from dividing the screen into smaller blocks. This is due to the fact that LZWx works better as the input size increases, but by dividing the screen we produce data of sizes smaller than the size of the original block; hence, the c.f. will decrease. For small blocks, the LZWx method will not gather enough data about the input to be able to produce a high c.f.

8.2. Screen Division

Table 8.2 gives the total c.f. when the screen is cut into two or three equal parts then each part is compressed alone using all previous compression methods. The table shows that the total c.f. of FAX is not

Table 8.1. Compression factors of image pdraw3 taken as a whole and as 4 parts and using all methods

Method	4 parts	Whole	$\frac{\text{4 parts}}{\text{whole}}$
FAX	2.91	4.19	0.69
LZW	2.58	4.00	0.65
LZW+R8	3.35	5.76	0.58
LZW+R4	3.16	5.59	0.57
LZW1	2.53	3.95	0.64
LZW1+R8	3.60	6.32	0.57
LZW1+R4	3.41	6.32	0.54
LZW2	2.53	3.96	0.64
LZW2+R8	3.59	6.54	0.55
LZW2+R4	3.40	6.24	0.54
LZW3	2.64	4.08	0.65
LZW3+R8	3.70	6.97	0.53
LZW3+R4	3.62	6.68	0.54
LZW3+LZWB1	2.18	3.40	0.64
LZWB	2.11	3.39	0.62
LZWB1	2.18	3.44	0.63
LZWB2-A	2.15	3.42	0.63
LZWB2-B	2.29	3.52	0.65

Table 8.2. Compression factors of romtxt and doc6a taken as whole 2-part and 3-part figures using all methods

	ROMTXT				DOC6A			
Method	2 parts	3 parts	2 parts / total	3 parts / total	2 parts	3 parts	2 parts / total	3 parts / total
FAX	1.40	1.40	1.00	1.00	9.52	9.44	0.99	0.98
LZW	2.12	1.98	0.90	0.84	4.41	4.14	0.91	0.85
LZW+R8	3.75	3.36	0.85	0.76	5.53	5.08	0.88	0.81
LZW+R4	3.52	3.12	0.84	0.74	5.60	5.22	0.89	0.83
LZW1	1.97	1.85	0.91	0.86	5.41	5.07	0.93	0.87
LZW1+R8	5.01	4.34	0.78	0.67	5.55	5.23	0.93	0.87
LZW1+R4	4.71	3.96	0.75	0.63	5.81	5.37	0.91	0.85
LZW2	1.97	1.85	0.91	0.86	5.61	5.17	0.95	0.88
LZW2+R8	5.01	4.34	0.78	0.67	5.60	5.10	0.92	0.84
LZW2+R4	4.66	3.98	0.76	0.65	5.81	5.36	0.93	0.86
LZW3	2.02	1.88	0.91	0.85	5.93	5.54	0.94	0.88
LZW3+R8	5.03	4.34	0.79	0.68	6.28	5.78	0.91	0.84
LZW3+R4	4.69	4.05	0.75	0.65	6.38	5.94	0.92	0.86
LZW3+LZWB1	1.43	1.43	0.91	0.85	5.60	5.36	0.94	0.90
LZWB	1.55	1.45	0.91	0.85	5.42	5.10	0.93	0.87
LZWB1	1.57	1.46	0.91	0.84	5.46	5.20	0.94	0.89
LZWB2-A	1.56	1.45	0.91	0.85	5.54	5.26	0.94	0.89
LZWB2-B	1.61	1.51	0.93	0.87	5.79	5.56	0.94	0.90

affected by this division while the total c.f. of LZWx is reduced by this division. This observation of FAX can be explained by the fact that FAX uses the information of only the previous line when coding the current line. This understanding of FAX allows us to assume that the total c.f. of compressing two or more screens together using FAX is, in fact, the same as the total c.f. when each screen is compressed alone. In the previous chapter, we implicitly used this result. Of course, LZWx benefits from compressing two or more screens together as was shown by the results of BIG in the previous chapter.

8.3. The Significance of the Groups Averages

Since there is no standard test to compare different compression algorithms, we developed the image data base described in Chapter 3. Comparing two compression methods based on the result of only one image or one group of images can be misleading. We avoid this problem by looking at the results of each group, the average of each group, and the average of all groups averages. This comprehensive checking makes sure that we avoid any anomaly that might exist in any image or group. But this creates another problem that might not be apparently noticeable; this problem is that this group averaging makes it subtle to notice the power these methods have when compressing some of the images. So, the best way is to use the group average and the average of all groups averages while keeping in mind that for some individual images (or groups) we may get a c.f. considerably higher than the average value. For the above reasons, we include Tables 8.3-8.22. Tables 8.3-8.10 contain the results

Table 8.3. Results of compressing images in group 1 using method R8+LZW2

Image	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
doc1a	7.10	8.56	1.37	1757	0
doc1b	3.63	23.13	1.48	3191	0
doc1c	13.85	4.78	1.31	1025	0
doc2a	8.77	8.19	1.32	1471	0
doc2b	7.69	9.12	1.27	1642	0
doc2c	11.44	6.09	1.26	1187	0
doc4a	7.33	11.80	1.43	1710	0
doc4b	6.77	14.44	1.43	1831	0
doc4c	5.21	4.88	0.66	1156	0
doc51a	4.58	11.26	1.10	2116	0
doc51b	7.70	7.64	1.09	1363	0
doc51c	10.93	2.53	0.60	692	0
doc5ra	4.83	11.09	1.04	1910	0
doc5rb	6.64	6.43	0.99	1460	0
doc5rc	4.55	4.73	0.61	1239	0
doc6a	6.07	12.09	1.32	2011	0
doc6b	11.35	6.32	1.32	1195	0
doc8	7.71	9.72	1.32	1638	0

Table 8.4. Results of compressing images in group 2 using method R8+LZW2

Image	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
frnch3a	7.23	10.22	1.37	1731	0
flowchrt	5.46	14.83	1.37	2208	0
electrc	4.70	18.23	1.37	2524	0
ordrfrm	7.98	10.16	1.43	1591	0
frnch1a	6.05	10.71	1.32	2018	0
doc2a	8.77	8.24	1.37	1471	0
doc2b	7.69	9.17	1.32	1642	0
AVERAGE	6.84	11.65	1.36	1884	0

Table 8.5. Results of compressing images in group 3 using method R8+LZW2

Image	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
romtxt	6.46	14.00	1.48	1907	0
frnch2a	3.41	29.88	1.42	3387	0
page1	8.75	9.34	1.37	1474	0
doc1-2	10.89	7.41	1.43	1234	0
cprog	13.85	5.00	1.32	1025	0
doc1b	3.63	23.07	1.42	3191	0
doc4a	7.33	11.81	1.48	1710	0
doc4b	6.77	14.45	1.43	1970	0
AVERAGE	7.64	14.37	1.42	1970	0

Table 8.6. Results of compressing images in group 4 using method R8+LZW2

File name	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
pdraw3	7.31	10.10	1.38	1715	0
science1	3.80	22.19	1.37	3063	0
science2	2.92	32.19	1.42	3910	0
doc51a	4.58	11.43	1.10	2116	0
AVERAGE	4.65	18.98	1.32	2701	0

Table 8.7. Results of compressing images in group 1 using method R4+LZW2

Image	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
doc1a	7.40	8.24	1.37	697	0
doc1b	3.68	22.90	1.48	3150	0
doc1c	14.39	4.89	1.32	996	0
doc2a	8.95	8.40	1.32	1447	0
doc2b	7.82	9.06	1.32	1619	0
doc2c	11.45	6.37	1.31	1186	0
doc4a	6.65	12.42	1.43	1858	0
doc4b	6.46	14.77	1.48	1907	0
doc4c	4.89	4.89	0.66	1214	0
doc51a	6.06	8.40	1.15	1663	0
doc51b	7.50	7.14	1.10	1392	0
doc51c	10.93	2.14	0.61	692	0
doc5ra	4.56	12.03	1.10	2009	0
doc5rb	6.45	6.98	0.99	1495	0
doc5rc	3.94	5.33	0.60	1392	0
doc6a	6.24	11.48	1.32	1965	0
doc6b	11.89	5.44	1.32	1152	0
doc8	8.26	9.44	1.32	1546	0
AVERAGE	7.64	8.91	1.18	1521	0

Table 8.8. Results of compressing images in group 2 using method R4+LZW2

Image	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
frnch3a	7.13	10.05	1.32	1750	0
flowchrt	5.60	13.35	1.37	2158	0
electrc	4.56	18.90	1.43	2593	0
ordrfrm	8.36	8.84	1.42	1530	0
frnchla	5.96	10.71	1.43	2045	0
doc2a	8.95	8.40	1.32	1447	0
doc2b	7.82	9.06	1.32	1619	0
AVERAGE	6.91	11.33	1.37	1877	0

Table 8.9. Results of compressing images in group 3 using method R4+LZW2

Image	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
romtxt	6.13	14.34	1.48	1996	0
frnch2a	2.69	40.20	1.54	4096	122
pagel	8.06	9.11	1.37	1579	0
docl-2	9.76	7.80	1.48	1348	0
cprog	12.16	5.38	1.43	1132	0
doclb	3.68	22.90	1.49	3150	0
doc4a	6.65	12.52	1.43	1858	0
doc4b	6.46	14.78	1.53	1907	0
AVERAGE	6.95	15.88	1.47	2133	15

Table 8.10. Results of compressing images in group 4 using method R4+LZW2

Image	Comprs. factor	Comprs. time s	Decomprs. time s	Table size	Extra calls
pdraw3	7.21	11.20	1.37	1735	0
science1	3.98	20.93	1.37	2932	0
science2	3.22	27.68	1.42	3567	0
doc51a	6.06	8.29	1.09	1663	0
AVERAGE	5.12	17.03	1.31	2474	0

Table 8.11. Results of compressing images in group 1 using method LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
doc1a	6.97	1.60	1786	0
doc1b	3.59	1.65	3228	0
doc1c	15.17	3.25	958	0
doc2a	7.96	1.54	1594	0
doc2b	7.32	1.53	1712	0
doc2c	10.78	1.54	1244	0
doc4a	2.77	1.54	4096	0
doc4b	2.35	1.59	4096	701
doc4c	2.17	0.72	2418	0
doc51a	3.92	1.26	2434	0
doc51b	6.04	1.21	1668	0
doc51c	12.35	0.66	652	0
doc5ra	2.86	1.26	3054	0
doc5rb	6.64	1.16	1460	0
doc5rc	3.11	0.66	1736	0
doc6a	6.30	1.59	1948	0
doc6b	11.68	1.53	1168	0
doc8	7.09	1.53	1760	0
AVERAGE	6.62	1.43	2056	39

Table 8.12. Results of compressing images in group 2 using method LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
frnch3a	7.06	1.54	1766	0
flowchrt	4.63	1.59	2558	0
electrc	4.25	1.60	2762	0
ordrfrm	5.38	1.59	2236	0
frnchla	6.25	1.59	1962	0
doc2a	7.96	1.54	1594	0
doc2b	7.32	1.54	1712	0
AVERAGE	6.12	1.57	2084	0

Table 8.13. Results of compressing images in group 3 using method LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
romtxt	2.22	1.65	4096	970
frnch2a	2.74	1.60	4096	51
pagel	3.99	1.64	2926	0
docl-2	4.61	1.54	2566	0
cprog	7.62	1.54	1654	0
doclb	3.59	1.59	3228	0
doc4a	2.77	1.54	4096	0
doc4b	2.35	1.64	4096	701
AVERAGE	3.74	1.59	3345	216

Table 8.14. Results of compressing images in group 4 using method LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
pdraw3	4.71	1.48	2520	0
science1	3.70	1.59	3136	0
science2	2.74	1.59	4096	56
doc51a	3.92	1.27	2434	0
AVERAGE	3.77	1.48	3047	14

Table 8.15. Results of compressing images in group 1 using method R8+LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
doc1a	7.35	2.08	1707	0
doc1b	3.81	1.59	3056	0
doc1c	15.15	3.35	959	0
doc2a	9.49	1.59	1379	0
doc2b	8.35	1.59	1532	0
doc2c	12.21	1.59	1128	0
doc4a	7.29	1.54	1718	7
doc4b	6.99	1.59	1780	0
doc4c	5.20	0.72	1158	0
doc51a	4.83	1.31	2022	0
doc51b	8.36	1.27	1276	0
doc51c	12.38	0.72	641	0
doc5ra	5.06	1.15	1836	0
doc5rb	6.94	1.15	1408	0
doc5rc	4.85	0.66	1179	0
doc6a	6.87	1.59	1808	0
doc6b	11.73	1.54	1164	0
doc8	8.39	1.60	1526	0
AVERAGE	8.07	1.48	1515	0

Table 8.16. Results of compressing images in group 2 using method R8+LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
frnch3a	7.83	1.65	1617	0
flowchrt	5.79	1.53	2098	0
electrc	4.99	1.49	2392	0
ordrfrm	8.60	1.54	1495	0
frnch1a	6.27	1.59	1956	0
doc2a	9.49	1.59	1379	0
doc2b	8.35	1.59	1532	0
AVERAGE	7.33	1.57	1781	0

Table 8.17. Results of compressing images in group 3 using method R8+LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
romtxt	6.40	1.53	1922	0
frnch2a	3.58	1.59	3238	0
page1	9.41	1.60	1389	0
doc1-2	11.49	1.59	1183	0
cprog	13.76	1.59	1030	0
doc1b	3.81	1.64	3056	0
doc4a	7.29	1.54	1718	0
doc4b	6.99	1.54	1780	0
AVERAGE	7.84	1.58	1915	0

Table 8.18. Results of compressing images in group 4 using method R8+LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
pdraw3	7.95	1.54	1597	0
science1	3.94	1.53	2962	0
science2	3.00	1.70	3814	0
doc51a	4.83	1.27	2022	0
AVERAGE	4.93	1.51	2599	0

Table 8.19. Results of compressing images in group 1 using method R4+LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
doc1a	7.48	2.42	1681	0
doc1b	3.86	2.47	3018	0
doc1c	15.12	3.68	960	0
doc2a	9.65	2.37	1360	0
doc2b	8.49	2.41	1511	0
doc2c	12.19	2.42	1130	0
doc4a	6.84	2.36	1814	0
doc4b	6.48	2.37	1902	0
doc4c	5.01	1.04	1191	0
doc51a	6.39	1.98	1590	0
doc51b	8.15	1.92	1302	0
doc51c	12.85	1.10	627	0
doc5ra	4.69	1.81	1960	0
doc5rb	7.06	1.82	1388	0
doc5rc	4.13	1.05	1339	0
doc6a	6.92	2.41	1797	0
doc6b	12.96	2.42	1078	0
doc8	8.84	2.42	1461	0
AVERAGE	8.17	2.14	1506	0

Table 8.20. Results of compressing images in group 2 using method R4+LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
frnch3a	7.96	2.37	1594	0
flowchrt	6.03	2.42	2025	0
electrc	4.91	2.36	2429	0
ordrfrm	8.83	2.41	1462	0
frnch1a	6.01	2.42	2030	0
doc2a	9.65	2.42	1360	0
doc2b	8.49	2.41	1511	0
AVERAGE	7.41	2.40	1773	0

Table 8.21. Results of compressing images in group 3 using method R4+LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
romtxt	6.25	2.36	1963	0
frnch2a	2.79	2.47	4073	0
page1	8.62	2.42	1493	0
doc1-2	10.48	2.41	1273	0
cprog	13.17	2.42	1065	0
doc1b	3.86	2.41	3018	0
doc4a	6.84	2.36	1814	0
doc4b	6.48	2.41	1902	0
AVERAGE	7.31	2.41	2075	0

Table 8.22. Results of compressing images in group 4 using method R4+LZW3

Image	Comprs. factor	Decomprs. time s	Table size	Extra calls
pdraw3	7.81	2.42	1620	0
science1	4.13	2.36	2840	0
science2	3.29	2.41	3496	0
doc51a	6.39	1.92	1590	0

of compressing g1, g2, g3, and g4 using Ry+LZW2. Tables 8.11-8.22 contain the results of compressing the above groups using LZW3 and Ry+LZW3. We have chosen these tables to show the detailed results of compressing each image or group using any LZWx method. Specifically, LZW3 was chosen because it has the highest c.f. among all LZWx methods and LZW2 was chosen because it is close to LZW1.

To illustrate the above points, we give the following examples:

1) The average c.f. of Table 8.22, which contains the results of compressing g4 using R4+LZW3, is bigger than the average c.f. of Table 8.18, which contains the results of compressing g4 using R8+LZW3; but the c.f. of the image pdraw3 in Table 8.18 is bigger than its c.f. in Table 8.22.

2) From Table 8.17, the average c.f. when compressing the images in g3 using R8+LZW3 is 7.84 whereas the c.f. of image doc1-2 is 11.49, i.e., considerably higher than the average c.f.

3) Tables 7.7 and 7.8 give the group averages using R8+LZW3 and R4+LZW3. From these tables, we see that R4+LZW3 gives higher groups average but R8+LZW3 gives higher c.f. for g3. Chapter 6 went into more detailed comparison of the groups results using Ry+LZWx.

8.4. Using the CCITT Documents for Comparison

To help in comparing the different methods we present Table 8.23 which contains the results of the total compression factors of images doc1, doc2, doc4, doc5, and doc6, where docx means docxa+docxb+...etc. Since these documents represent typical documents, it is easier to compare the methods using Table 8.23. Comparing the methods using this table, we get:

1) For doc1, R4+LZW3 has the highest c.f. among the other LZW methods. This c.f., 6.54, is slightly less than the

Table 8.23. Compression factors of the CCITT standard documents using all methods

Method	Documents					Average
	doc1	doc2	doc4	doc5	doc6	
FAX	6.67	14.89	1.69	4.74	12.00	8.00
LZW	6.03	7.97	2.66	4.32	5.67	5.33
LZW+R8	6.07	9.34	4.89	5.38	7.55	6.65
LZW+R4	6.05	9.35	4.52	5.37	7.82	6.62
LZW1	5.95	7.90	2.40	4.31	7.74	5.66
LZW1+R8	6.16	8.95	6.64	5.78	7.74	7.05
LZW1+R4	6.21	9.03	6.13	5.94	8.09	7.08
LZW2	5.95	7.96	2.14	4.32	7.81	5.64
LZW2+R8	6.14	9.05	6.62	5.83	7.91	7.11
LZW2+R4	6.30	9.18	6.17	5.95	8.18	7.16
LZW3	6.15	8.45	2.47	4.49	8.19	5.95
LZW3+R8	6.46	9.77	6.69	6.20	8.67	7.56
LZW3+R4	6.54	9.89	6.28	6.34	9.02	7.61
LZW3+LZWB1	5.79	8.05	1.90	3.82	7.72	5.46
LZWB	5.79	8.01	1.95	3.80	7.45	5.40
LZWB1	5.82	7.99	1.97	3.80	7.42	5.40
LZWB2-A	5.90	8.13	1.96	3.85	7.59	5.49
LZWB2-B	6.18	8.58	1.99	4.01	7.91	5.73

c.f. of FAX, 6.67. FAX did better because the image contains a lot of empty spaces.

2) For doc2, R4+LZW3 has the highest c.f., 9.89, among the other LZW methods. The c.f. of FAX is 50% higher. FAX did much better than R4+LZW3 because the image is a very simple graphics screen with long black runs and short white runs.

3) For doc4, R8+LZW3 has the highest c.f. among the other LZW methods and the ratio of this c.f. to the corresponding c.f. of FAX is 3.96. This ratio is too high because doc4 contains only textual data; and as we showed before Ry+LZWx does extremely better than FAX for textual data.

4) For doc5, R4+LZW3 has the highest c.f. among the other LZW methods and the ratio of this c.f. to the corresponding c.f. of FAX is 1.34. The ratio is higher than 1 because the screen contains textual data. The fact that doc5 contains both text and graphics explains why the ratio is not as high as in the case of doc4. R4+LZW3 has higher c.f. than R8+LZW3 in this case due to the effect of the graphics data in doc5.

5) For doc6, R4+LZW3 has the highest c.f. among the other LZW methods. This c.f. is 75% of the c.f. of FAX. The reason that FAX has the highest c.f. is that doc6 is any easy graphics screen. doc6 is not an easy graphics screen as doc2 is; this explains the difference between the ratio of the c.f. of R4+LZW3 to that of FAX for doc6 and the same ratio for doc2. This shows that as the graphics get more complex R4+LZWx becomes better till it produces a c.f. higher than FAX.

6) We note that among the LZW methods, R4+LZW3 has the highest c.f. for graphics screens and screens that have both textual and graphics data. R8+LZW3 has the highest c.f. for textual screens.

7) Points 1 to 6 above agree with the observations we found in Chapter 7.

8) Among all the LZW methods, R4+LZW3 has the highest average of the 5 images c.f. The average in the case of FAX was higher because of the high c.f. that FAX has for doc2 and doc6.

9) The c.f. of Ry+LZW3 is close to the c.f. of Ry+LZW1 and Ry+LZW2. The c.f. of Ry+LZW3 is bigger by no more than 10%. A similar trend is observed when the c.f. of LZW3 is compared to the c.f. of LZW1 and LZW2.

8.5. Results of Group 5

In Chapter 4, we presented the results of compressing the graphics blocks in g5. In Chapter 5, we presented the corresponding results using LZW. The results of LZW show that LZW do not produce a c.f. higher than the c.f. of FAX for g5. The tables for the groups averages using all the LZW methods agree with this. This result agrees with the observation we mentioned before in Section 8.1 that the LZW c.f. will decrease if the image is divided into smaller blocks. Hence, in the results of the modifications on LZW, we do not give a table for g5; instead, we only give the averages of each group.

8.6. Results of Group 8

In Chapter 3, group 8 was introduced to test the power of each method. To help in comparing the results of these methods when compressing the images in g8, we included the c.f. for all the methods in Table 8.24. From this table, we observe the following:

1) For images blok6, boxes, and lines LZW3+LZWB1 gives the highest c.f. among all the methods, including FAX. This shows that the concept of the LZWBs is optimum for this kind of data. It also shows the need to use different varieties of true images, as we did in the image data base, to compare the methods because, as we showed in Chapter 6, LZW3+LZWB1 did not perform as good as it is performing here.

2) The c.f. of Ry+LZW1 or Ry+ZLW2 are close to the c.f. of Ry+LZW3. Similarly, LZW1 and LZW2 give c.f. close to the

Table 8.24. Compression factors of group 8 using all methods

Method	Image					
	blok3	blok6	boxes	lines	test1	usamap
FAX	109.03	24.54	69.57	32.07	49.60	1.56
LZW	27.97	10.98	16.06	15.19	12.79	6.57
LZW+R8	15.47	11.80	21.30	15.41	15.01	7.64
LZW+R4	16.79	14.00	23.39	16.68	14.61	7.27
LZW1	51.78	60.61	58.18	81.22	26.19	6.36
LZW1+R8	36.87	24.84	38.37	52.81	18.99	7.33
LZW1+R4	37.91	27.35	44.94	55.75	21.84	7.51
LZW2	56.74	63.49	65.04	82.47	30.49	6.37
LZW2+R8	41.99	25.44	54.05	54.05	20.22	7.36
LZW2+R4	37.12	31.94	43.36	54.98	22.75	7.26
LZW3	60.15	66.12	68.67	101.27	29.88	6.82
LZW3+R8	49.38	28.07	55.75	65.84	27.65	8.26
LZW3+R4	46.11	42.11	59.93	73.06	25.54	7.93
LZW3+LZWB1	70.18	91.95	75.47	137.93	30.29	6.47
LZWB	54.05	24.69	30.36	38.37	22.60	NA
LZWB1	46.11	26.53	29.47	37.65	21.61	5.68
LZWB2-A	54.05	25.04	30.36	38.37	23.94	NA
LZWB2-B	24.92	30.36	30.36	38.37	23.94	NA

c.f. of LZW3. This shows that although LZW1 or LZW2 are not the optimum LZWx, they are close to the optimum method LZW3 without its complexity.

3) The image "usamap" is an example where FAX fails to take advantage of the redundancy present in some images. The redundancy of this image is in the interior of the map which consists of strings of 0101... etc. that represent the filling of the map. LZWx was able to detect this redundancy and give a higher c.f. R8+LZW3 has the highest c.f. for usamap, namely, 8.26. The ratio of the c.f. of R8+LZW3 to that of FAX is 5.29.

8.7. The Significance of "Extracalls"

The method LZW has a maximum number of symbols that it can recognize; this number is the table maximum size. The compressor and decompressor agree not to put more symbols in the table if the table is filled up. This means that the LZW method loses its adaptability to the new input if the table is filled up. To measure the effect of filling up the table on the compression process, we count the number of the unsuccessful calls to the table after the table is filled up; this number is the variable "extracalls" in the results of LZW and the modified LZWs.

In the results of LZW, and its modifications, the extracalls were averaged for each group. This average value is misleading most of the time since most of the images do not require extracalls but the average shows that they do. So, the average of extracalls is meaningful only if compressing each image in a group requires extra calls.

8.8. Table Size

The methods LZWx assume the maximum size of the LZW table to be 4096, which requires 12 bits to represent each symbol. But the results show that, for some images and groups, the number of symbols that are actually used is considerably less than the table maximum size. Using this fact, we propose to limit the size of the table for the images or groups that use symbols less than the table maximum size. By limiting the table size, we limit the length of each symbol, decrease the size of the output of LZWx, and, hence, increase the c.f. For example, if we let the table maximum size be 1024, the length of each symbol is only 10 bits; for an image that has a table size less than 1024 the c.f., will increase by exactly 20%, ((12/10)-1)100. This table size limitation is not arbitrary if we use a fixed addressing or a fixed symbol length scheme, which we will. In the case of a fixed length symbol, the table size must be only a number that is a power of 2 since any other number will result in losing some symbols. For example, if the maximum table size was chosen to be 2000, LZW needs 12 bits to address or represent each symbol. But if we use 12 bits, we can represent up to 4096 symbols. So, this 12-bit length of the symbol allows us to use the symbols 2001 to 4096 which we will lose if we choose the maximum size to be 2000 symbols.

From the results of Ry+LZWx, we find the following:

1) For all Ry+LZWx, the average table size of g5 never exceeded 1024. Hence, the table size of compressing g5 can be limited to 1024 giving an approximately 20% increase in the c.f. The increase is approximate because some of

the images in g5 require more than 1024 symbols.

2) For all Ry+LZWx, the average table size of g1 and g2 is less than 2048. Hence, the table size of compressing g1 and g2 can be limited to 2048 giving an approximately 9% increase in the c.f.

3) For R8+LZWx ($x = 1$, 2, and 3), the average table size of g3 is less than 2048. Hence, as in the above point, the table maximum size can be set to 2048.

4) For doc4a and romtxt using R8+LZW2, the tables size are 1710 and 1907, respectively, and the c.f. are 7.33 and 6.46, respectively. If we let the table maximum size be 2048, the c.f. of doc4a and romtxt will be 8 and 7.05, respectively. These new c.f. are very close to the c.f. if the image was sent as an ASCII code. This is an important result because it shows that, as we mentioned in last chapter, we can get a c.f. very close to and sometimes better than the c.f. of pattern recognition without worrying about the difficulties of pattern recognition.

It should be noted that the way the code for the LZWx was written makes it easy to change the code in order to let the table maximum size be adaptive but no more than 4096.

8.9. Remarks about R8 and R4

R8 and R4 were designed with the assumption that it is easy to find the characters' height and then divide the screen accordingly; nevertheless, it was envisioned that even if this information is not known, these two methods will still give a high c.f. The image frnch2a proves our vision because, although the image is in a textual format that is different than the one R8 and R4 was designed for, the ratio of the resulted c.f. to the c.f. when using FAX is 1.77 which is a considerable increase.

Finding the height of the text lines is a matter that can be easily solved. In fact, in some of the pattern recognition techniques, finding the height of each character is one feature, among many features, that should be extracted. Refer to [11] and [8].

9. CONCLUSION

In this work, the author developed a number of new improved compression algorithms, an extended test data base, an analysis of library needs, and a variety of test results. From this work, a number of conclusions were drawn as enumerated below.

1) For easy graphics images, i.e., images containing long runs of black pels and short runs of white pels, FAX gives high c.f. that is satisfactory to the goal of this research. For textual screens and complex graphics FAX performs poorly.

2) The LZW method was simulated and gave a c.f. better than that of FAX for the images for which FAX did poorly. But LZW was not as good as FAX for the easy graphics images.

3) Three new methods, that use the fact that the input to LZW is a long string of pels of a scanned screen, were proposed and investigated. The first method, LZWB, counts the run-lengths of the screen and sends them to LZW. The second method, LZWB1, uses part of the run-lengths used in the first method and adds to them codes for some of the most probable two and three runs. The third method, LZWB2, counts the run-lengths as in the first method; in addition to that, it initializes the LZW table with some of the most probable two and three run-lengths. Each of these proposed methods showed an improvement in the c.f. It was explained that in the case of colored images, it would be expected from these methods to give a better c.f.

4) An improvement, LZW3, in LZW, as suggested in [43], was simulated, and, in general, a gave c.f. higher than LZW. LZW3 needs long c.t., so we proposed two versions that avoid the long searches required by LZW3. These two proposals, LZW1 and LZW2, give c.f. close to that of LZW but much shorter c.t.

5) Two improvements in the way LZWs scan the screen were suggested. These improvements, R8 and R4, work with any of the above LZWs. They produced higher c.f. than when using the LZWs alone and even in some cases gave smaller d.t.

6) Combining two or three images in the compression using Ry+LZWx (for x = 0, 2, 3) was investigated and, in general, produced a higher compression factor than compressing each image alone.

7) The library survey that was presented in Chapter 3 showed that about 50% of the library material was in text format. The detailed format of the text varies from one library material, e.g., a book or a magazine, to another.

8) Using some of the proposed methods, e.g., R8+LZW2, it was possible to reach a c.f. for a textual screen that is close to or even higher than the c.f. of compression methods that employ a pattern recognition technique. The proposed methods are much simpler to implement, need much less computation, and are more adaptive to the data change.

From the above observations, we reach the conclusion that R8+LZW3 should be used unless we are compressing a screen that is full of easy graphics. In this case of easy graphics screens, the system should be able to compress the screen using FAX and inform the receiver of the change in the compression method. The library system can handle the long c.t. of R8+LZW3. The d.t. of R8+LZW3, which is in the range of 1 to 2 s, is acceptable for the library system. The c.t. of LZW3 is higher than that of other LZWx methods but, as was mentioned at the beginning of the research, the compression in the library system is done once so the c.t. is allowed to be long. For real time compression, LZW1 or LZW2 should be used instead of LZW3.

The system should also be able to detect the needed maximum size of the table and signal the receiver accordingly.

9.1. Suggestions for Future Work

The following points are suggested and should be investigated:

1) The maximum size of the LZW table should be increased over 4096 to compress many images at the same time or to compress colored images. Increasing the table size increases the c.t., d.t., and, hopefully, the c.f. Long c.t. is tolerable in the library system. Since both the d.t. and the c.f. increase at the same time, there is a trade-off that needs to be investigated.

2) The modifications of LZWBs to work on colored images.

3) The use of method BIG to compress an actual page which usually consists of more than one screen.

4) The success of LZW for this type of data indicates that more methods in the field of data compression via textual substitution should be investigated as image compression methods.

5) LZW builds its table using the first character that has not been sent yet. This gives LZW a look-ahead feature that raises its c.f. The methods LZW1, LZW2, and LZW3 do not have this look-ahead feature so their d.t. is shorter than LZW, but this feature may raise their c.f., specifically for textual screens. So, a modified LZW1-LZW3 that include the look-ahead feature should be investigated.

6) The application of LZWx in more than one pass that may increase the c.f. This may be better than increasing the table size.

7) Implementing the LZWx in hardware. [43] reported on a hardware implementation but with no details.

8) The use of Ry+LZWx for library material images captured using a camera or a scanner. The c.f. obtained in this thesis using FAX for the screen images are much smaller than the values reported for images scanned at high resolution and compressed using FAX. So, the c.f. for scanned documents using Ry+LZWx should be investigated.

9) Applying Ry+LZWx to images other than library material like astronomical and medical images.

10) Changing the FAX modified Huffman table, although we think it will not be beneficial as we induced before.

11) Improving FAX so that it can use the information from lines before the previous line in order to code the current line, and from parts other than the parts next to each other.

12) Compressing the output of FAX, after modifying this output, using any of the LZWx methods.

13) The extension of both FAX and Ry+LZWx to colored images.

14) Developing a method similar to R4 but whose block height is only 4 pels. Developing similar methods with different block height.

15) Using a hashing function to speed up the search in the LZW table in order to decrease the c.t. Examples of simple hashing functions are the following:

    a) The number of characters, and not symbols, in the string.

    b) The count of the values of the characters in the string.

    c) The third character in the string.

    For the kind of strings we get in the LZW table while compressing the library material images, it is envisioned that any of these simple functions will perform successfully.

10. REFERENCES

1. Costigan, D. M. Electronic Delivery of Documents and Graphics. New York: Van Nostrand Reinhold Company, 1978.

2. Rosenheck, B. M. "Fastfax, a second generation facsimile system employing redundancy reduction techniques." IEEE Transactions on Communication Technology, 18, No. 6 (December 1970):772-779.

3. Netravali, A. N., F. W. Mounts, and E. G. Bowen. "Ordering techniques for coding of two-tone facsimile pictures." The Bell System Technical Journal, 55, No. 10 (December 1976):1539-1552.

4. Musmann, H. G., and D. Preuss. "Comparison of redundancy reducing codes for facsimile transmission of documents." IEEE Transactions on Communications, 25, No. 11 (November 1877):1425-1433.

5. Mounts, F. W., E. G. Bowen, and A. N. Netravali. "An ordering scheme for facsimile coding." The Bell System Technical Journal, 58, No. 9 (November 1979):2113-2128.

6. Huang, T. S. "Coding of two-tone images." IEEE Transactions on Communications, 25, No. 11 (November 1877):1406-1424.

7. Hartke, D. H., W. M. Sterking, and J. E. Shemer. "Design of a raster display processor for office applications." IEEE Transactions on Computers, 27, No. 4 (April 1978):337-348.

8. Ismail, M. G., and R. J. Clarke. "Adaptive block/location coding of facsimile signals using subsampling and interpolation for pre- and postprocessing." IEEE Transactions on Communications, 29, No. 12 (December 1981):1925-1933.

9. Yamada, T. "Edge-difference coding — a new, efficient redundancy reduction technique for facsimile signals." IEEE Transactions on Communications, 27, No. 8 (August 1979):1210-1217.

10. Costigan, D. M. "Facsimile comes up to speed." IEEE Communications Magazine, 18, No. 3 (May 1980):30-35.

11. Pratt, W. K., P. J. Capitant, W. H. Chen, E. R. Hamilton, and R. H. Wallis. "Combined symbol matching facsimile data compression system." Proceedings of the IEEE, 68, No. 7 (July 1980):786-796.

12. Yasuda, Y. "Overview of digital facsimile coding techniques in Japan." Proceedings of the IEEE, 68, No. 7 (July 1980):830-844.

13. Henter, R., and A. H. Robinson. "International digital facsimile coding standards." Proceedings of the IEEE, 68, No. 7 (July 1980): 854-867.

14. Doherty, B. "Comparison of facsimile data compression schemes." IEEE Transactions on Communications, 28, No. 12 (December 1980): 2023-2024.

15. Kim, J. K., and P. Segin. "A conditional incremental-runlength code based on two-dimensional Markov model." IEEE Transactions on Communications, 29, No. 10 (October 1981):1527-1532.

16. Johnson, O., and A. N. Netravali. "Progressive transmission of two-tone images." IEEE Transactions on Communications, 29, No. 12 (December 1981):1934-1941.

17. Netravali, A. N., and E. G. Bowen. "A picture browsing system." IEEE Transactions on Communications, 29, No. 12 (December 1981): 1968-1976.

18. Teramuka, H., K. Ono, S. Ando, Y. Yamazaki, S. Yamamato, and K. Matsuo. "Experimental facsimile communication system on packet switched data network." IEEE Transactions on Communications, 29, No. 12 (December 1981):1942-1951.

19. Matsumoto, M., H. Ochi, and S. Yoshino. "A high-speed facsimile apparatus for satellite communication." IEEE Transactions on Communications, 29, No. 12 (December 1981):1952-1958.

20. Bodson, D., and R. Schaphorst. "Error sensitivity of CCITT standard facsimile coding techniques." IEEE Transactions on Communications, 31, No. 1 (January 1983):69-81.

21. Silver, D. M., and D. A. H. Johnson. "Facsimile coding using symbol-matching techniques." Proceedings of the IEEE, 131, No. 2 (April 1984):125-129.

22. Johnson, O., J. Segen, and G. L. Cosh. "Coding of two-level pictures by pattern matching and substitution." The Bell System Technical Journal, 62, No. 8, Part 1 (October 1983):2513-2545.

23. Anderson, K. L., F. C. Mintzer, G. Geortzel, J. L. Mitchel, K. S. Pennington, and W. B. Pennebaker. "Binary-image-manipulation algorithms in the image view facility." IBM Journal of Research and Development, 31, No. 1 (January 1987):16-31.

24. Davisson, L. D. "Universal noiseless coding." IEEE Transactions on Information Theory, 19, No. 6 (November 1973):783-795.

25. Rissanen, J. "A universal data compression system." IEEE Transactions on Information Theory, 29, No. 5 (September 1973):576-664.

26. Rissanen, J. "Universal coding, information, prediction, and Estimation." IEEE Transactions on Information Theory, 30, No. 4 (July 1984):629-636.

27. Fitingof, B. M. "Optimal coding in the case of unknown and changing message statistics." Problems of Information Transmission, 2, No. 2 (Summer 1966):1-7.

28. Fitingof, B. M. "The compression of descrete information." Problems of Information Transmission, 3, No. 3 (Fall 1967):22-29.

29. Rissanen, Jorma, and Glen G. Langdon. "Universal modeling and coding." IEEE Transactions on Information Theory, 27, No. 1 (January 1981):12-23.

30. Krichevsky, Raphail E., and Victor K. Trofmov. "The performance of universal encoding." IEEE Transactions on Information Theory, 27, No. 2 (March 1981):199-207.

31. Davisson, Lee D., Robert J. McEliece, Michael B. Pursley, and Mark S. Wallace. "Efficient universal noiseless source code." IEEE Transactions on Information Theory, 27, No. 3 (May 1981):269-279.

32. Ziv, J. "Coding of sources with unknown statistics -- part I: Probability of encoding error." IEEE Transactions on Information Theory, 18, No. 3 (May 1972):384-389.

33. Lempel, A., and J. Ziv. "On the complexity of finite sequences." IEEE Transactions on Information Theory, 22, No. 1 (January 1976): 75-81.

34. Ziv, J., and A. Lempel. "A universal algorithm for data compression." IEEE Transactions on Information Theory, 23, No. 3 (May 1977):337-343.

35. Ziv, J. "Coding theorems for individual sequences." IEEE Transactions on Information Theory, 24, No. 4 (July 2978):405-412.

36. Ziv, J., and A. Lempel. "Compression of individual sequences via variable-rate coding." IEEE Transactions on Information Theory, 24, No. 5 (September 1978):530-536.

37. Rodeh, M., V. R. Pratt, and S. Even. "Linear algorithm for data compression via string matching." Journal of the Association for Computing Machinery, 28, No. 1 (January 1981):16-24.

38. Storer, J. A., and T. G. Szymanski. "Data compression via textual substitution." Journal of the Association for Computing Machinery, 29, No. 4 (October 1982):928-951.

39. Gilbert, E. N., and C. L. Monma. "Multigram codes." IEEE Transactions on Information Theory, 28, No. 2 (March 1982):346-348.

40. Longdon, G. G., Jr. "A note on the Ziv-Lempel model for compressing individual sequences." IEEE Transactions on Information Theory, 29, No. 2 (March 1983):284-287.

41. Welch, T. A. "A technique for high-performance data compression." Computer, 17, No. 6 (June 1984):8-19.

42. Mahlab, D. "The complexity of data compression algorithms." Section 2.1.1, pp. 1-4. In the 14th Convention of Electrical and Electronics Engineers in Israel Proceedings, Tel Aviv, 26-28 March 1985.

43. Greiss, I. "Adaptive data compression using self-synchronized parsing algorithm." Section 2.2.5, pp. 1-4. In The 14th Convention of Electrical and Electronics Engineers in Israel Proceedings, Tel Aviv, 26-28 March 1985.

44. Lempel, A., and J. Ziv. "Compression of two-dimensional data." IEEE Transactions on Information Theory, 32, No. 1 (January 1986): 2-8.

45. Shokely, James E. Introduction to number theory. New York: Holt, Rinehart and Winston, Inc., 1976.

11. ACKNOWLEDGMENTS

The author wishes to express his sincere appreciation to Dr. Arthur V. Pohm for his encouragement and guidance during the work leading to this thesis. The unlimited encouragement of Dr. Pohm helped the author overcome many difficulties and obstacles encountered during this research.

Dr. T. Smay suggested to the author to write the programs code in both C and assembly languages instead of assembly language alone. The author appreciates this suggestion that helped simplify the coding of these programs.

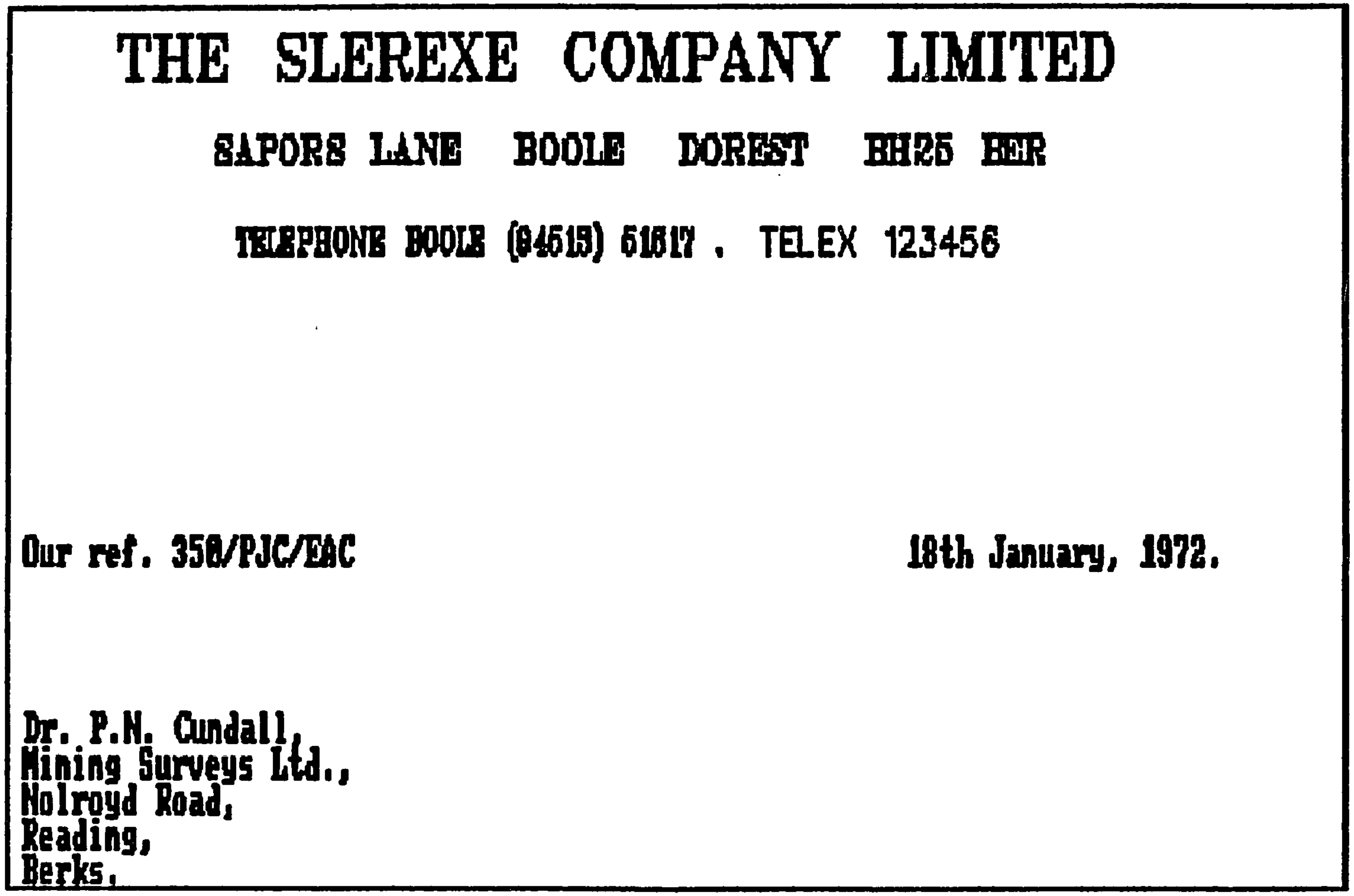
Acknowledgment must be made to the Saudi Arabian Embassy and the Saudi ambassador, Prince Bander Bin Sultan, for the grant that the author used to buy the hardware and software used in this work.

The author also wishes to thank Mr. Ahmad Sayes, one of the author's best friends, for helping him in debugging some of the programs and drawing many of the images that the author used in the data base.

The author would like also to thank Maggie Wheelock for typing the manuscript of this thesis.

Special thanks are due to my wife for her understanding, patience, and unfailing encouragement, and to my two sons Osama and Ziad for their love and patience; and to my lovely daughter Rooaa who, with her lively playing and smiling face, helped ease many of my difficult times.

12. APPENDIX A. IMAGES USED IN THE DATA BASE



THE SLEREXE COMPANY LIMITED

SAPORS LANE BOOLE DOREST BH25 8ER

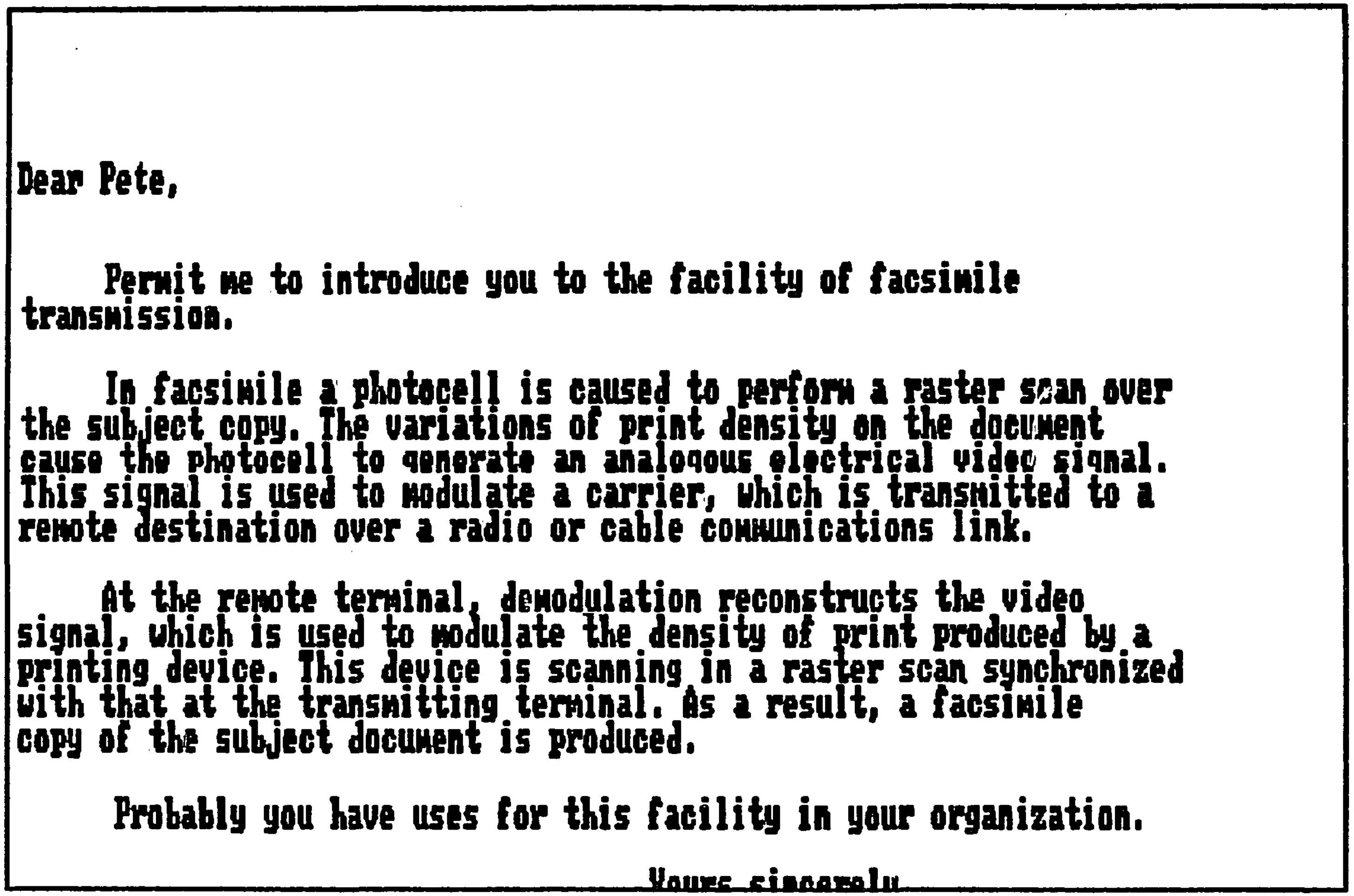
TELEPHONE BOOLE (94513) 51617 . TELEX 123456

Our ref. 350/PJC/EAC

18th January, 1972.

Dr. P.N. Cundall,
Mining Surveys Ltd.,
Holroyd Road,
Reading,
Berks.

Figure 12.1. Image docla



Dear Pete,

Permit me to introduce you to the facility of facsimile transmission.

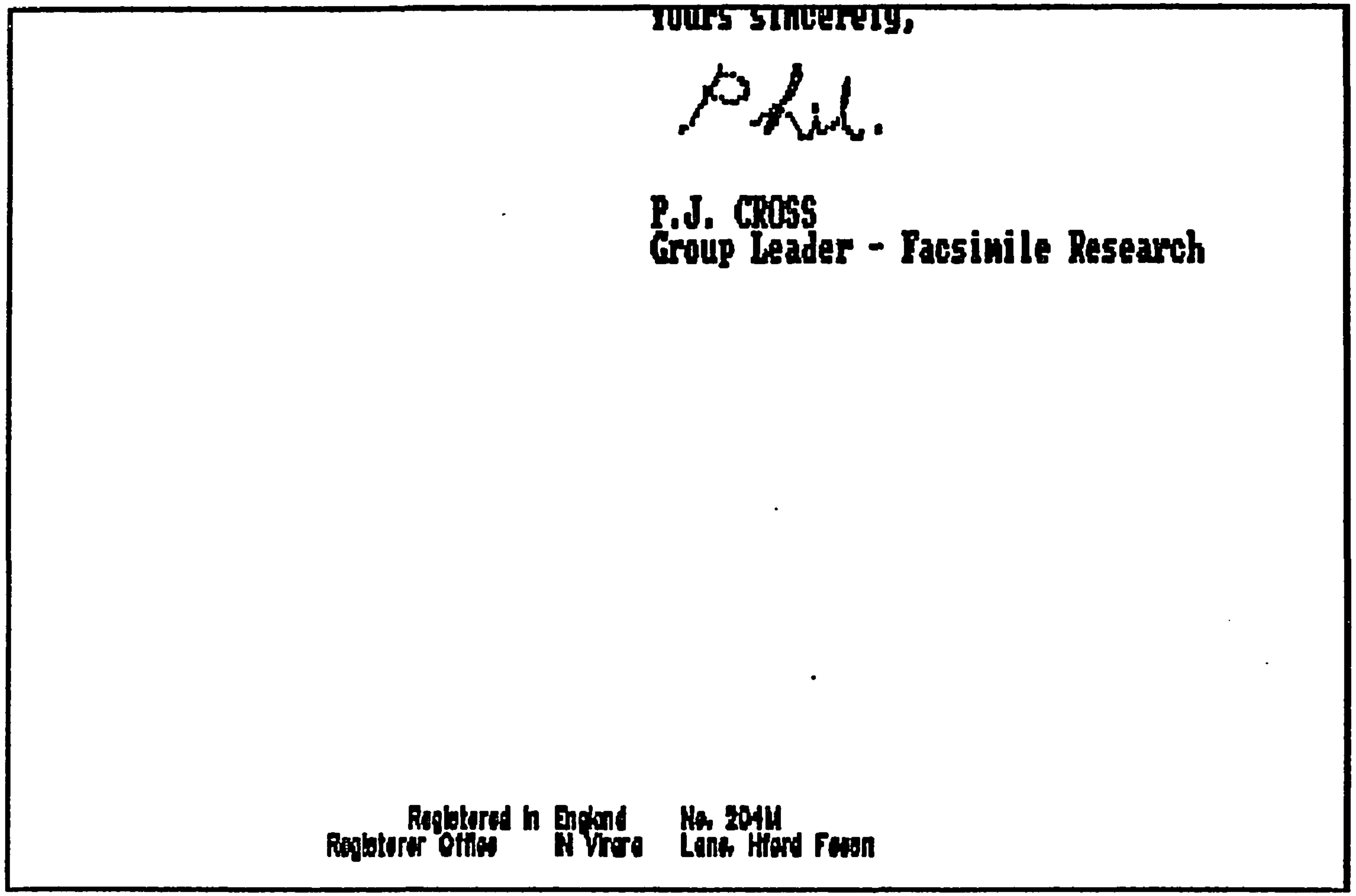
In facsimile a photocell is caused to perform a raster scan over the subject copy. The variations of print density on the document cause the photocell to generate an analogous electrical video signal. This signal is used to modulate a carrier, which is transmitted to a remote destination over a radio or cable communications link.

At the remote terminal, demodulation reconstructs the video signal, which is used to modulate the density of print produced by a printing device. This device is scanning in a raster scan synchronized with that at the transmitting terminal. As a result, a facsimile copy of the subject document is produced.

Probably you have uses for this facility in your organization.

Yours sincerely

Figure 12.2. Image doc1b



Yours sincerely,

Phil.

P.J. CROSS
Group Leader - Facsimile Research

Registered in England No. 204M
Registered Office N Virara Lane, Ilford Essex

Figure 12.3. Image doc1c

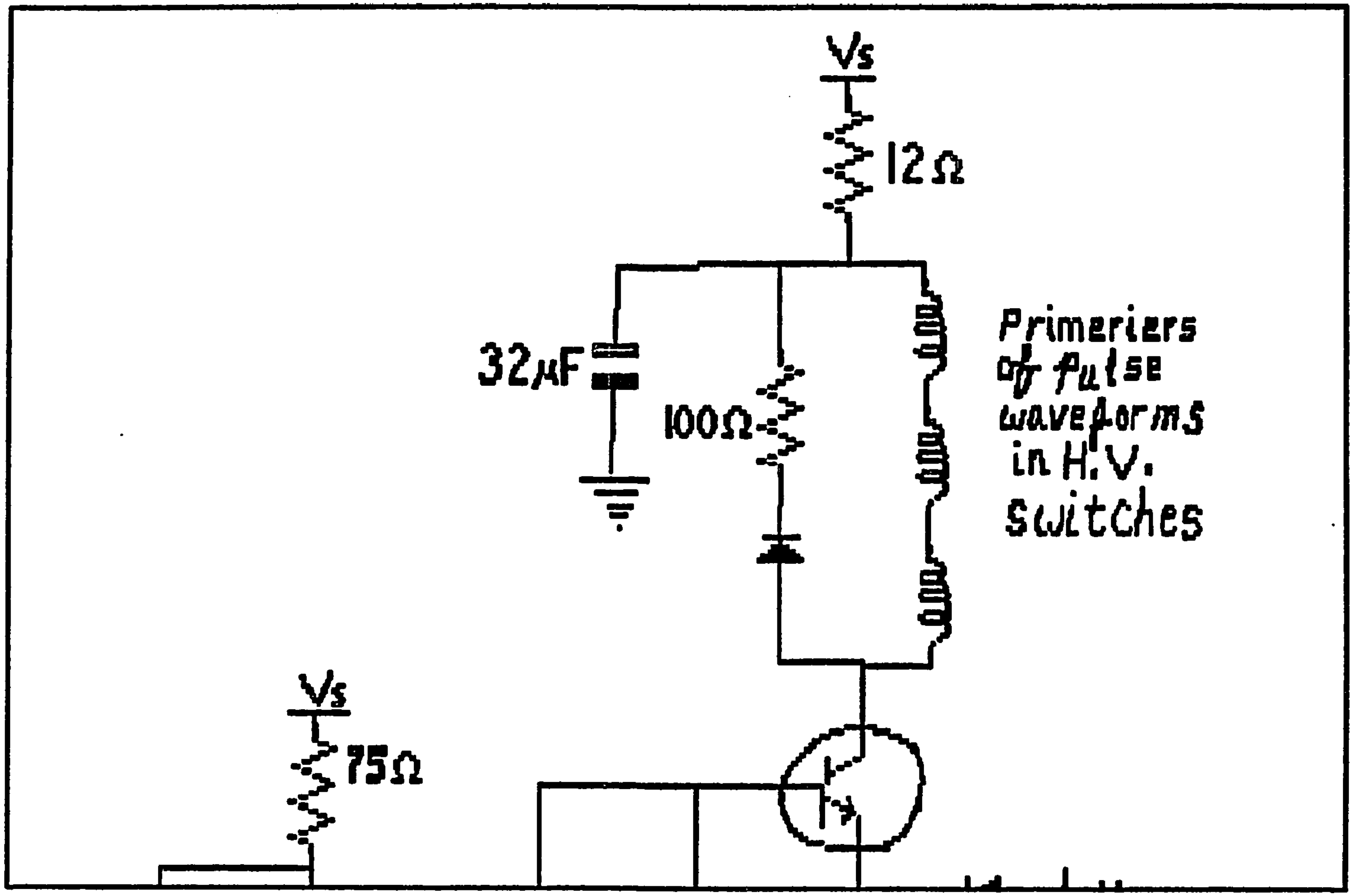


Figure 12.4. Image doc2a

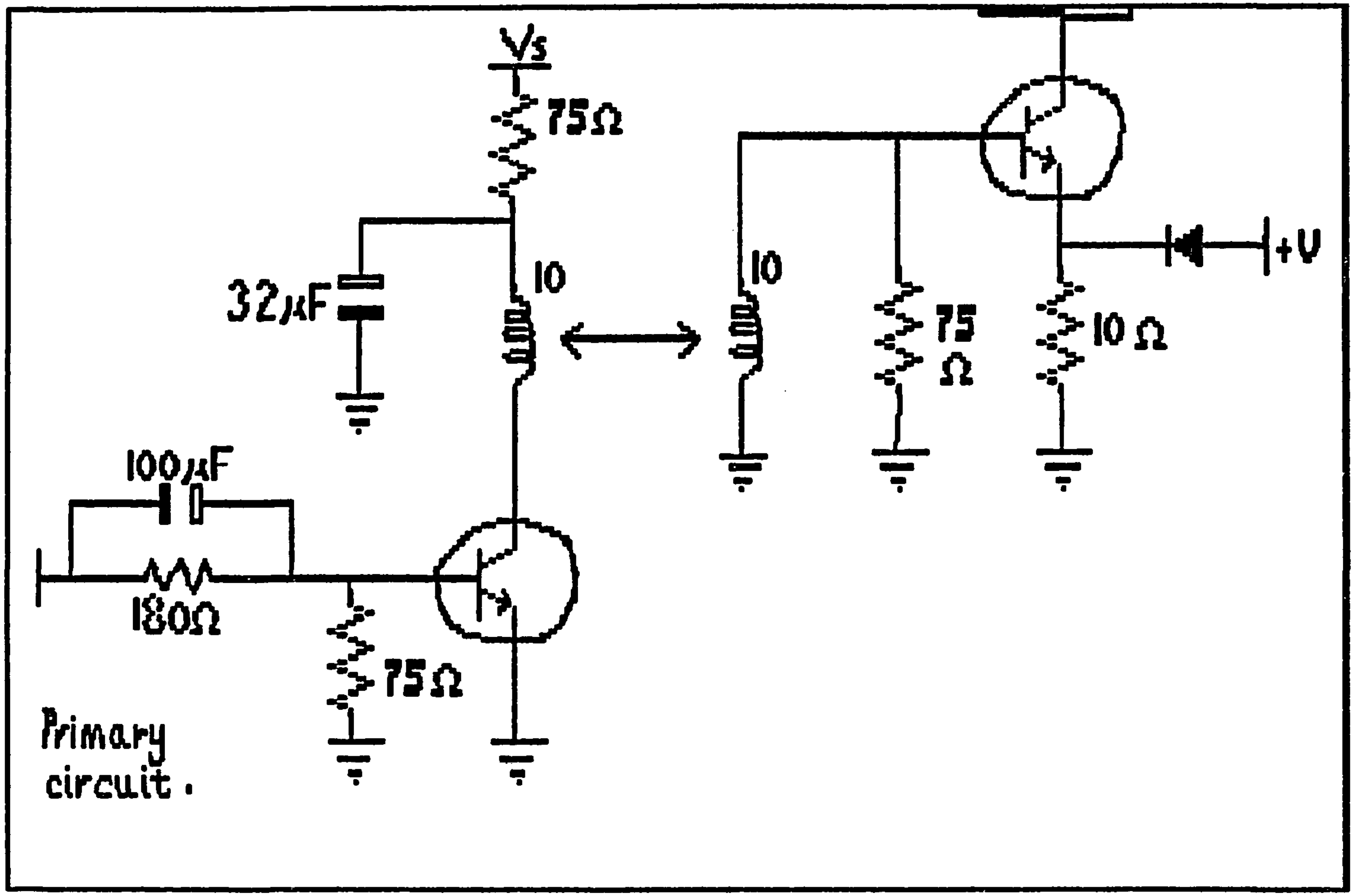
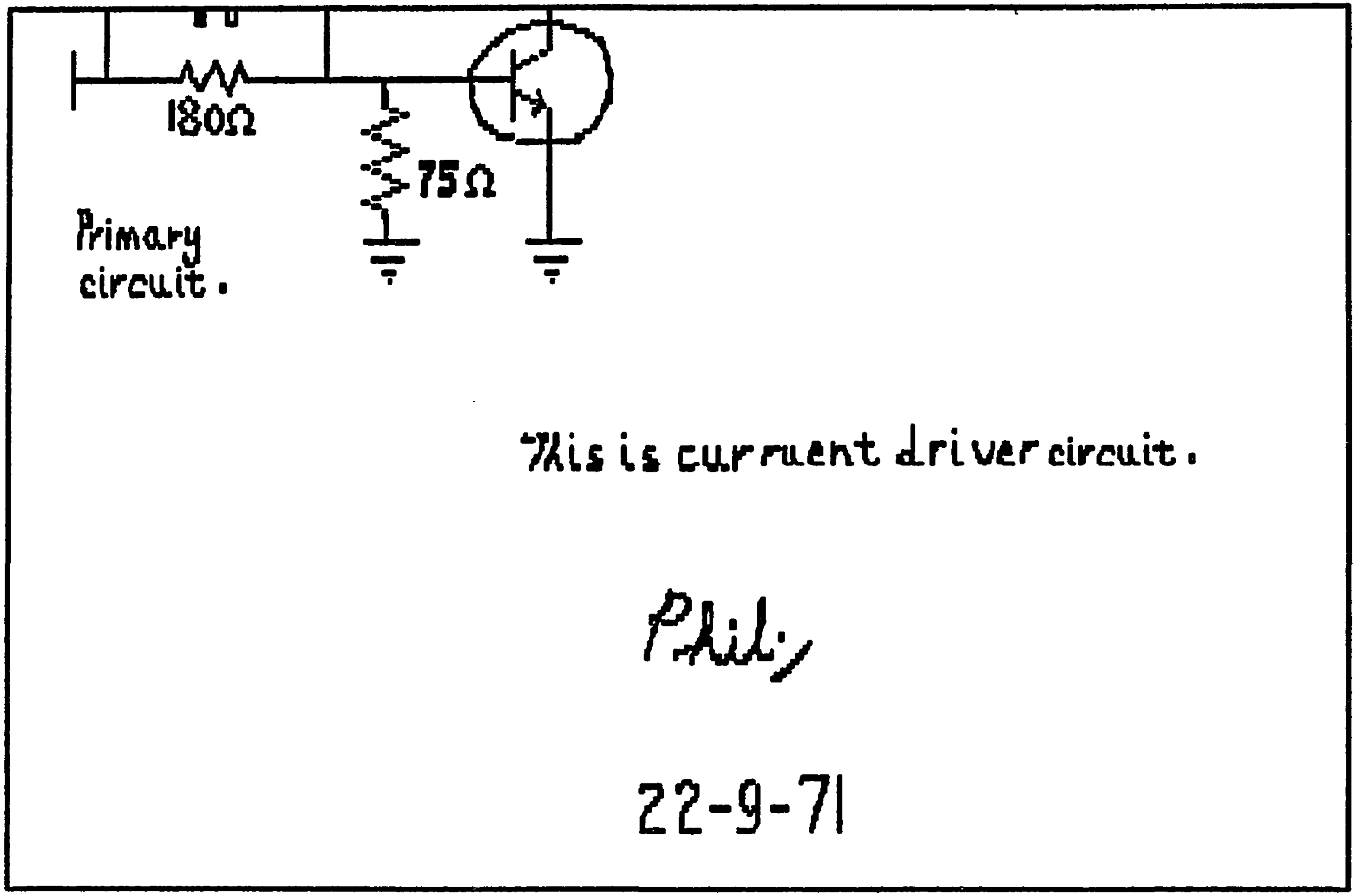


Figure 12.5. Image doc2b



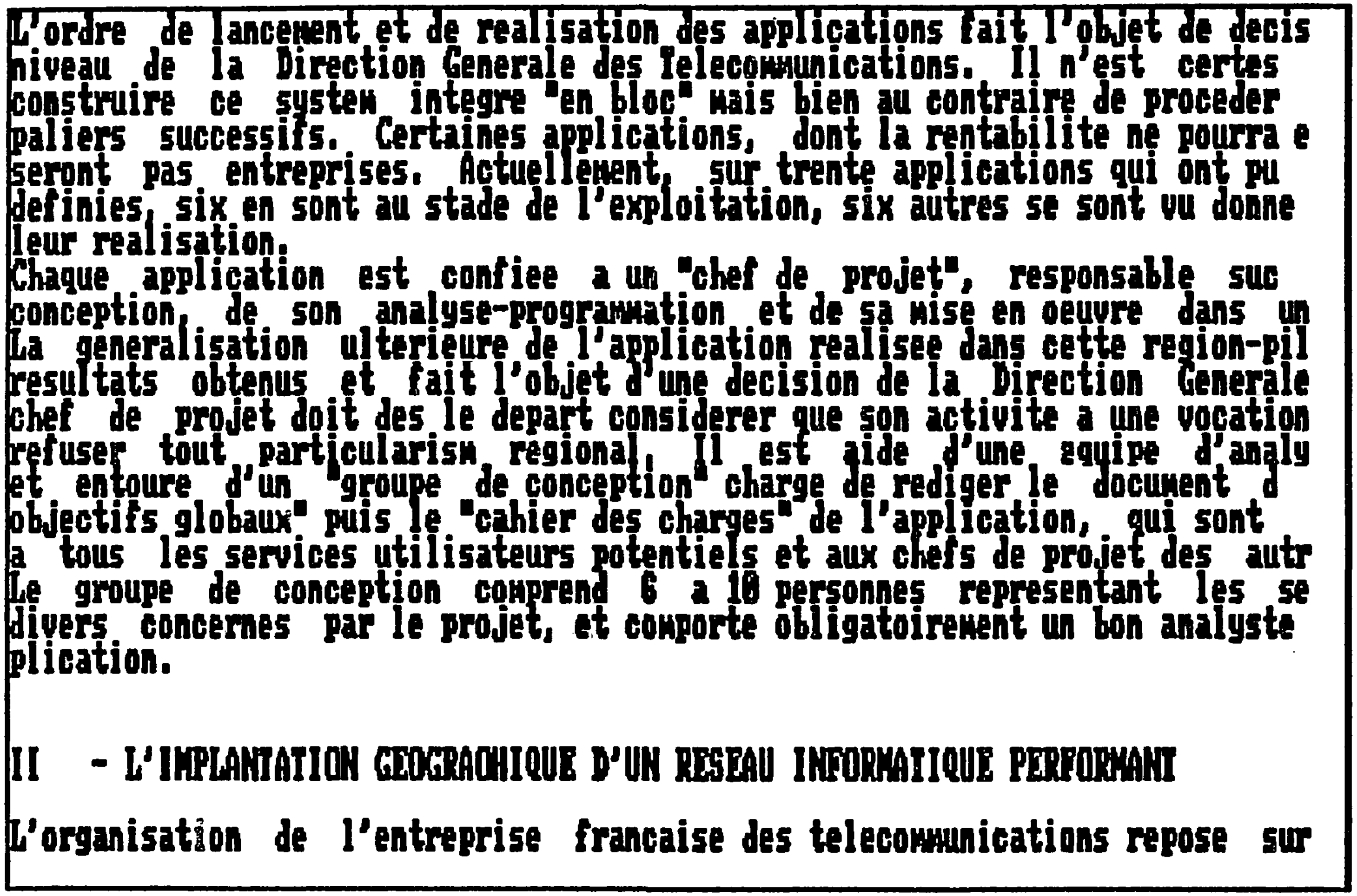
Primary circuit.

This is current driver circuit.

Phil

22-9-71

Figure 12.6. Image doc2c



L'ordre de lancement et de realisation des applications fait l'objet de decis
niveau de la Direction Generale des Telecommunications. Il n'est certes
construire ce system integre "en bloc" mais bien au contraire de proceder
paliers successifs. Certaines applications, dont la rentabilite ne pourra e
seront pas entreprises. Actuellement, sur trente applications qui ont pu
definies, six en sont au stade de l'exploitation, six autres se sont vu donne
leur realisation.
Chaque application est confiee a un "chef de projet", responsable suc
conception, de son analyse-programmation et de sa mise en oeuvre dans un
La generalisation ulterieure de l'application realisee dans cette region-pil
resultats obtenus et fait l'objet d'une decision de la Direction Generale
chef de projet doit des le depart considerer que son activite a une vocation
refuser tout particularism regional. Il est aide d'une equipe d'analy
et entoure d'un "groupe de conception" charge de rediger le document d
objectifs globaux" puis le "cahier des charges" de l'application, qui sont
a tous les services utilisateurs potentiels et aux chefs de projet des autr
Le groupe de conception comprend 6 a 10 personnes representant les se
divers concernes par le projet, et comporte obligatoirement un bon analyste
plication.

II - L'IMPLANTATION GEOGRAOHIQUE D'UN RESEAU INFORMATIQUE PERFORMANT

L'organisation de l'entreprise francaise des telecommunications repose sur

Figure 12.7. Image doc4a

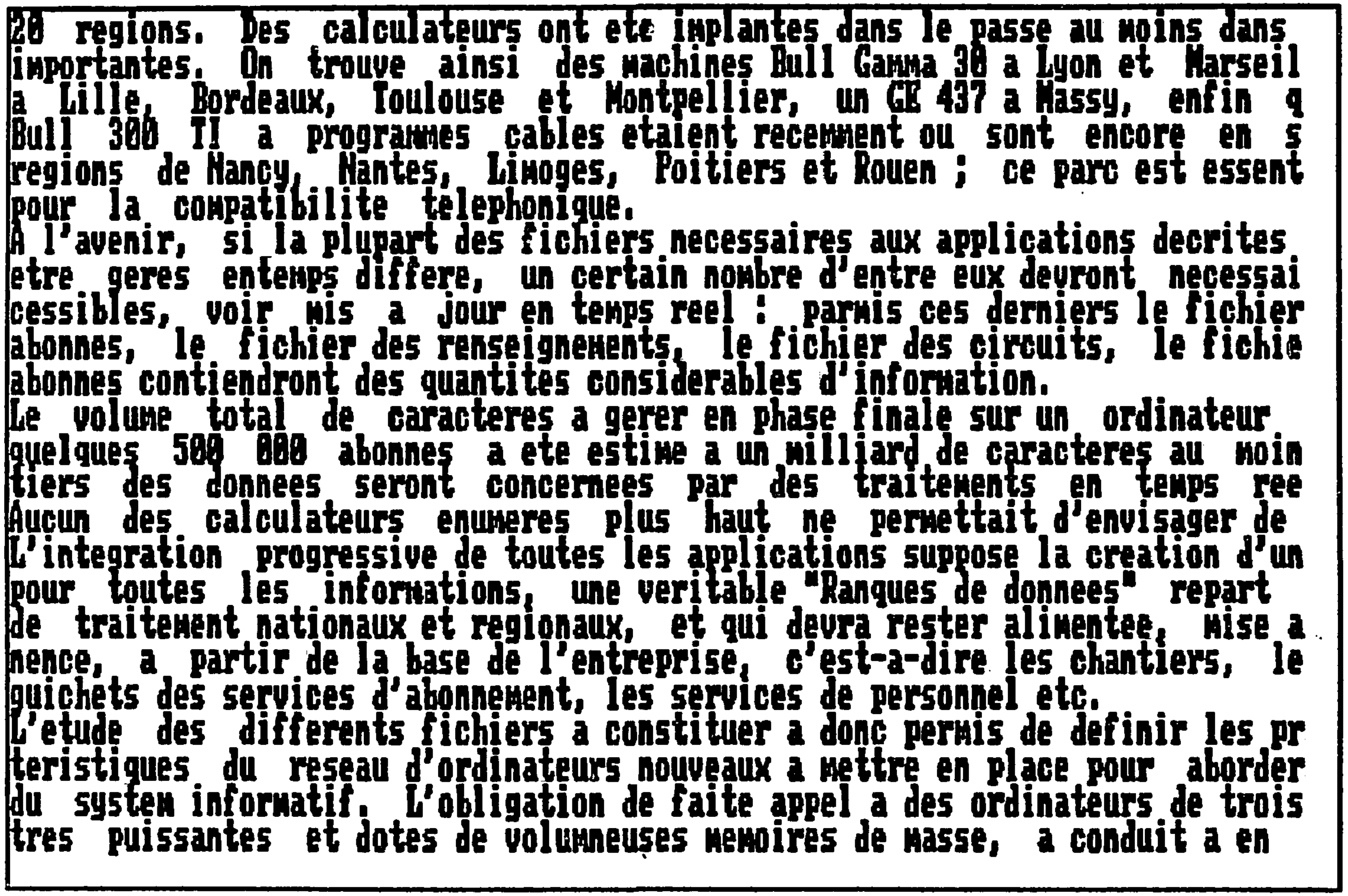
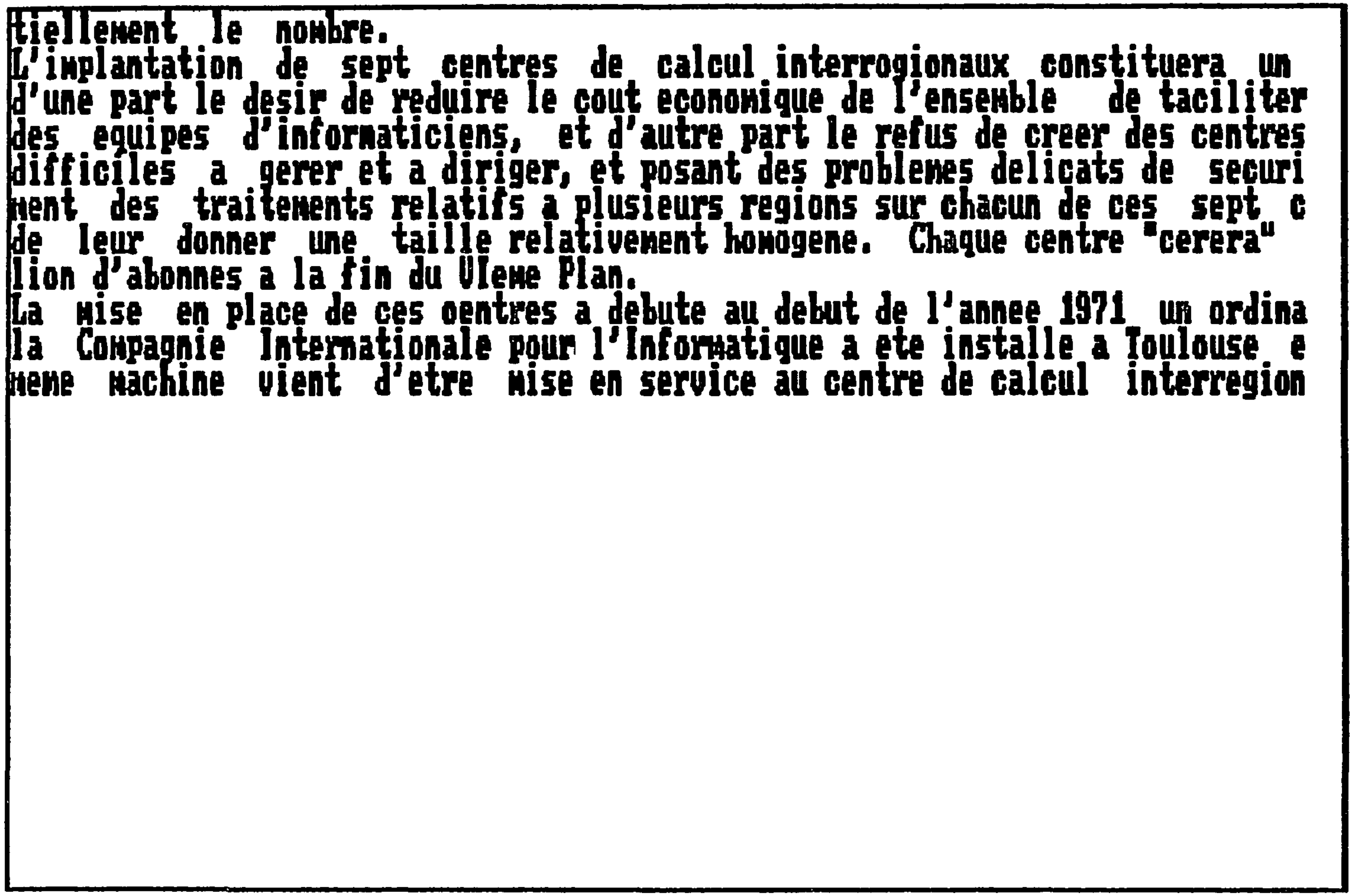
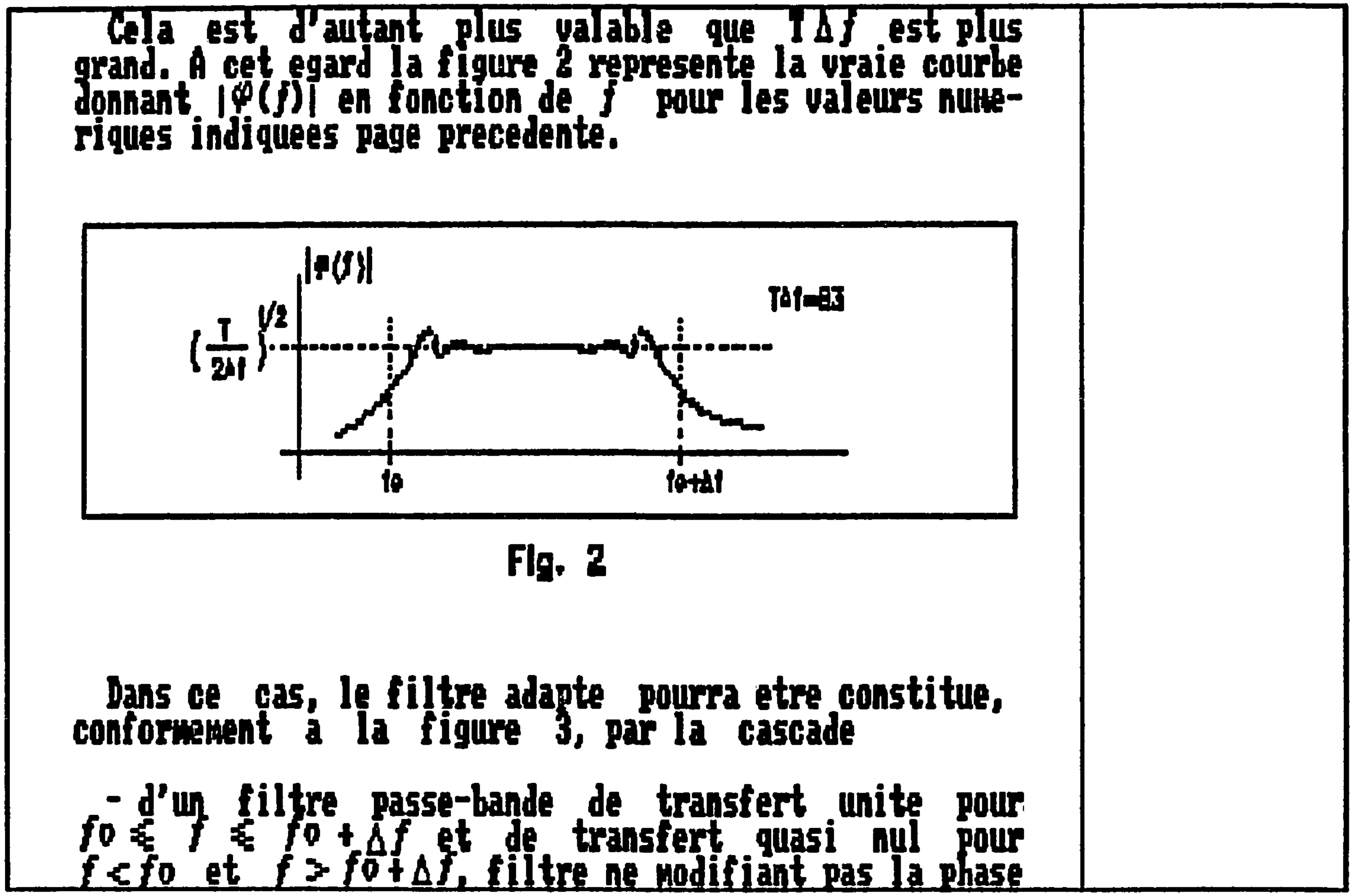

20 regions. Des calculateurs ont ete implantes dans le passe au moins dans
importantes. On trouve ainsi des machines Bull Gamma 30 a Lyon et Marseil
a Lille, Bordeaux, Toulouse et Montpellier, un GE 437 a Massy, enfin q
Bull 300 TI a programmes cables etaient recemment ou sont encore en s
regions de Nancy, Nantes, Limoges, Poitiers et Rouen ; ce parc est essent
pour la compatibilite telephonique.
A l'avenir, si la plupart des fichiers necessaires aux applications decrites
etre geres entemps differe, un certain nombre d'entre eux devront necessai
cessibles, voir mis a jour en temps reel : parmis ces derniers le fichier
abonnes, le fichier des renseignements, le fichier des circuits, le fichie
abonnes contiendront des quantites considerables d'information.
Le volume total de caracteres a gerer en phase finale sur un ordinateur
quelques 500 000 abonnes a ete estime a un milliard de caracteres au moin
tiers des donnees seront concernees par des traitements en temps ree
Aucun des calculateurs enumeres plus haut ne permettait d'envisager de
L'integration progressive de toutes les applications suppose la creation d'un
pour toutes les informations, une veritable "Banques de donnees" repart
de traitement nationaux et regionaux, et qui devra rester alimentee, mise a
nence, a partir de la base de l'entreprise, c'est-a-dire les chantiers, le
guichets des services d'abonnement, les services de personnel etc.
L'etude des differents fichiers a constituer a donc permis de definir les pr
teristiques du reseau d'ordinateurs nouveaux a mettre en place pour aborder
du system informatif. L'obligation de faite appel a des ordinateurs de trois
tres puissantes et dotes de volumneuses memoires de masse, a conduit a en

Figure 12.8. Image doc4b



tiellement le nombre.
L'implantation de sept centres de calcul interrogionaux constituera un
d'une part le desir de reduire le cout economique de l'ensemble de taciliter
des equipes d'informaticiens, et d'autre part le refus de creer des centres
difficiles a gerer et a diriger, et posant des problemes delicats de securi
ment des traitements relatifs a plusieurs regions sur chacun de ces sept c
de leur donner une taille relativement homogene. Chaque centre "cerera"
lion d'abonnes a la fin du VIeme Plan.
La mise en place de ces oentres a debute au debut de l'annee 1971 un ordina
la Compagnie Internationale pour l'Informatique a ete installe a Toulouse e
meme machine vient d'etre mise en service au centre de calcul interregion

Figure 12.9. Image doc4c



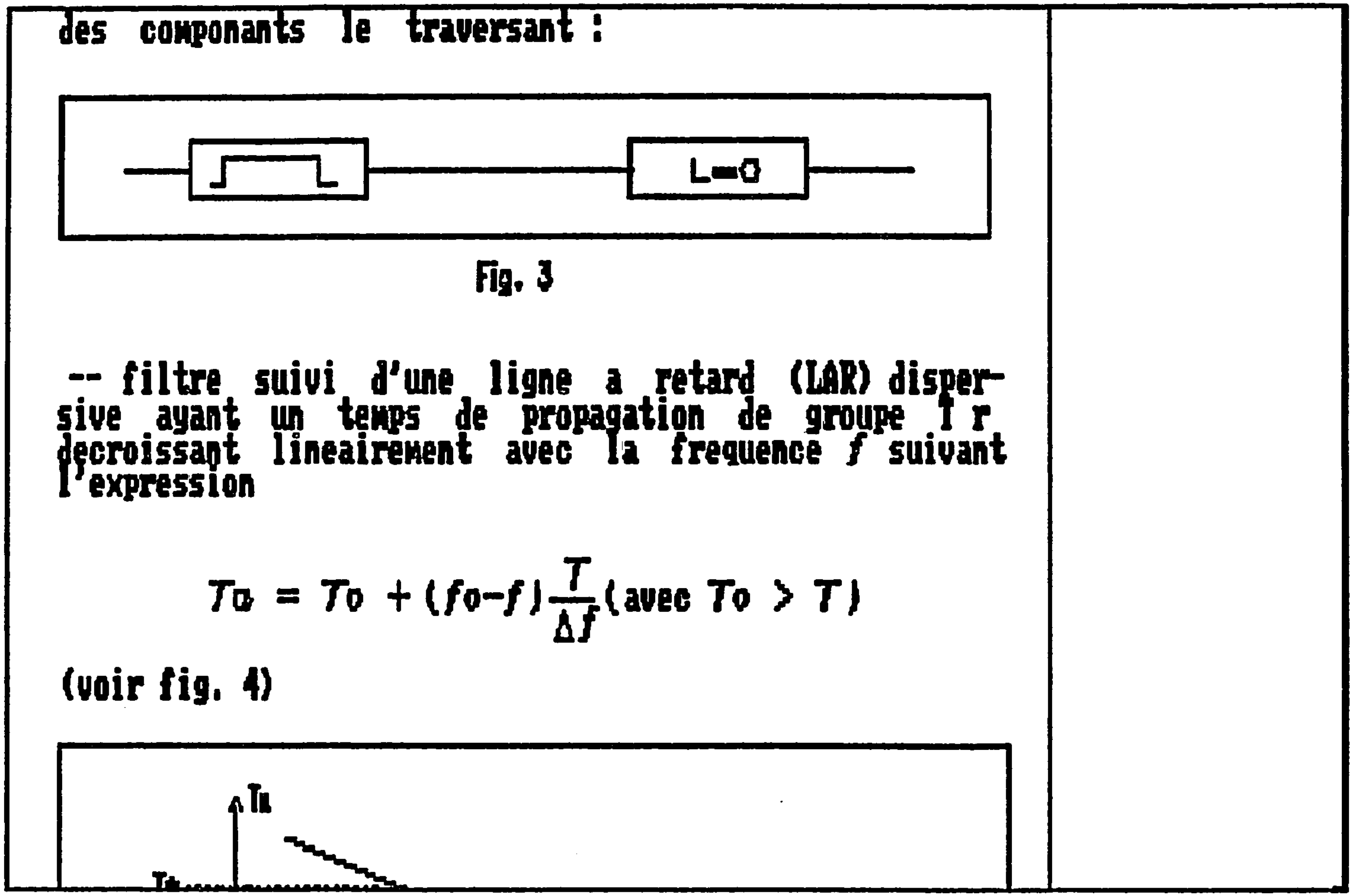
Cela est d'autant plus valable que $T\Delta f$ est plus grand. A cet egard la figure 2 represente la vraie courbe donnant $|\varphi(f)|$ en fonction de f pour les valeurs numeriques indiquees page precedente.

Fig. 2

Dans ce cas, le filtre adapte pourra etre constitue, conformement a la figure 3, par la cascade

- d'un filtre passe-bande de transfert unite pour $f_0 \leqslant f \leqslant f_0 + \Delta f$ et de transfert quasi nul pour $f < f_0$ et $f > f_0 + \Delta f$, filtre ne modifiant pas la phase

Figure 12.10. Image doc5la



des componants le traversant :

Fig. 3

-- filtre suivi d'une ligne a retard (LAR) dispersive ayant un temps de propagation de groupe T_r decroissant lineairement avec la frequence f suivant l'expression

$$T_{\alpha} = T_0 + (f_0 - f)\frac{T}{\Delta f} \text{(avec } T_0 > T\text{)}$$

(voir fig. 4)

Figure 12.11. Image doc51b

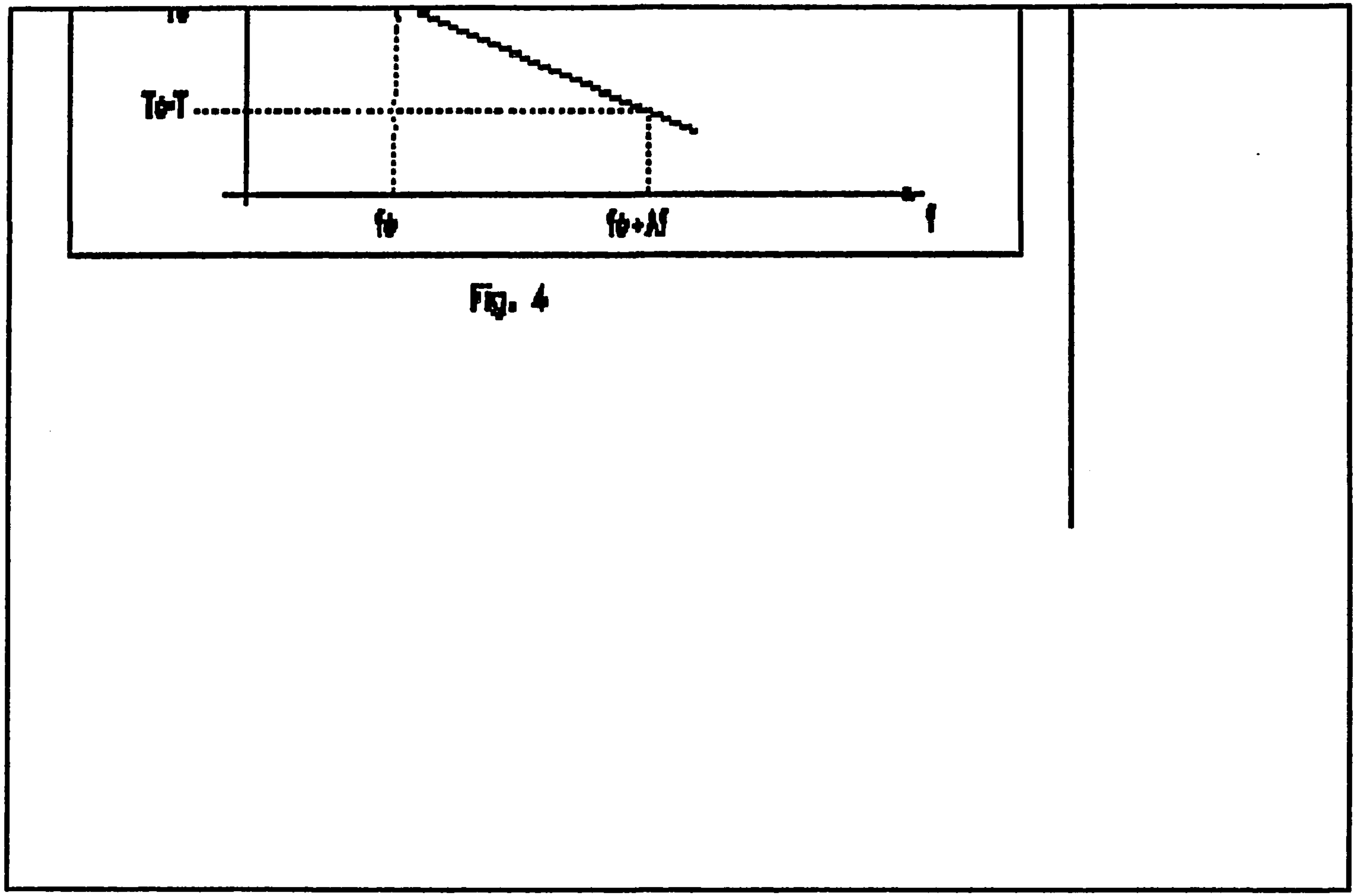
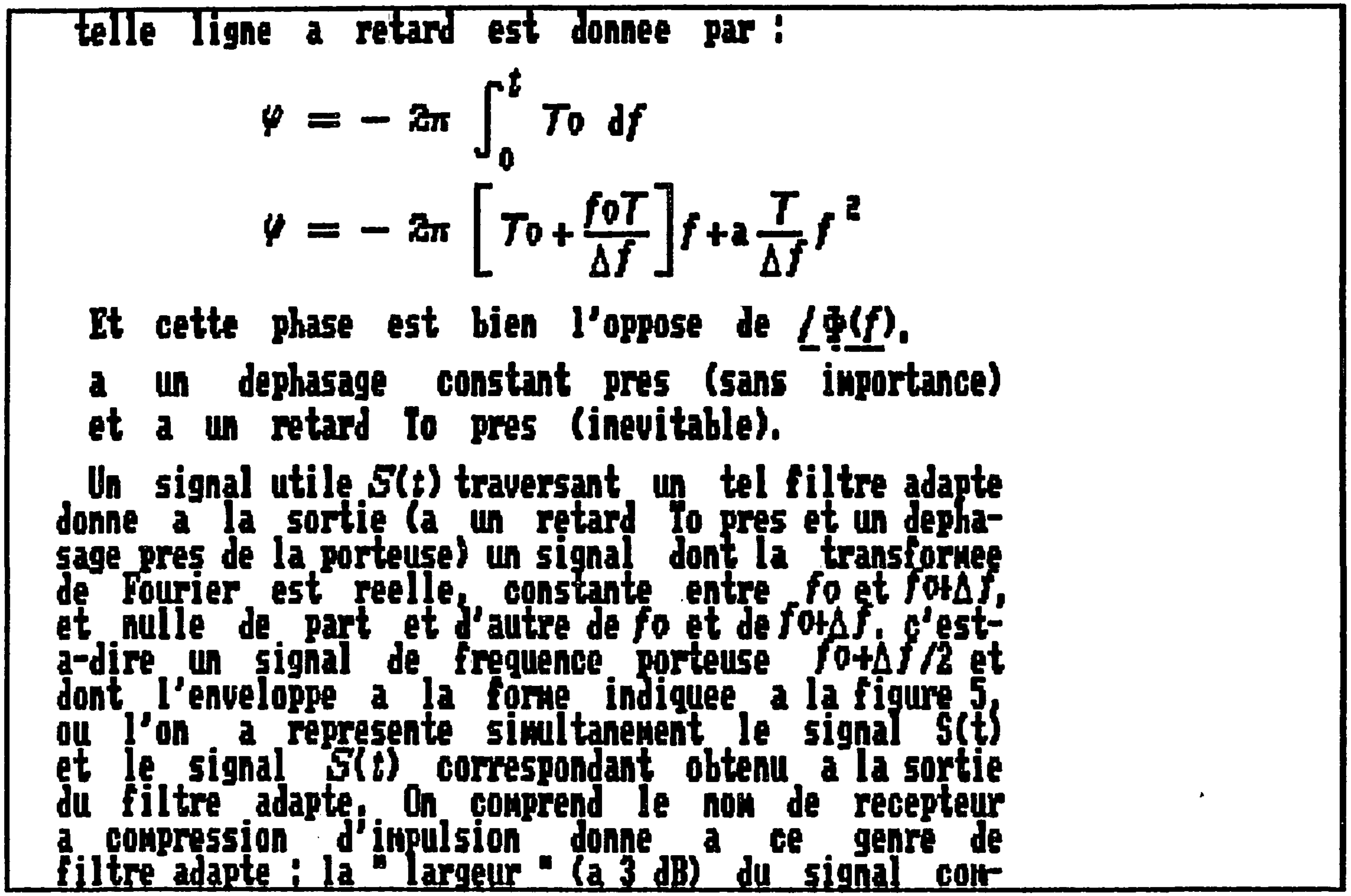


Figure 12.12. Image doc51c



telle ligne a retard est donnee par :

$$\psi = -2\pi \int_0^t To \; df$$

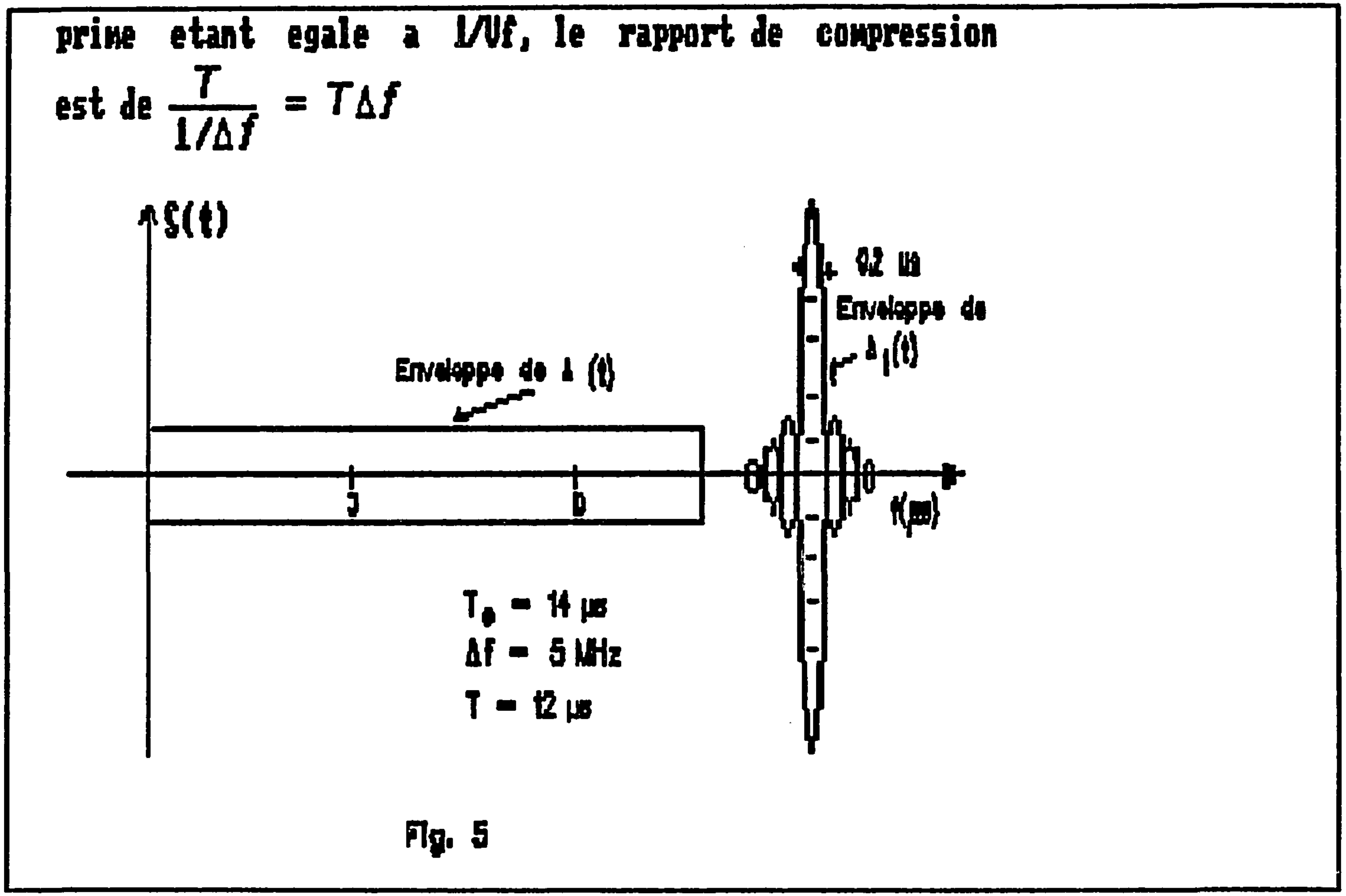
$$\psi = -2\pi \left[To + \frac{foT}{\Delta f} \right] f + a \frac{T}{\Delta f} f^2$$

Et cette phase est bien l'oppose de $\underline{/\Phi(f)}$.

a un dephasage constant pres (sans importance) et a un retard To pres (inevitable).

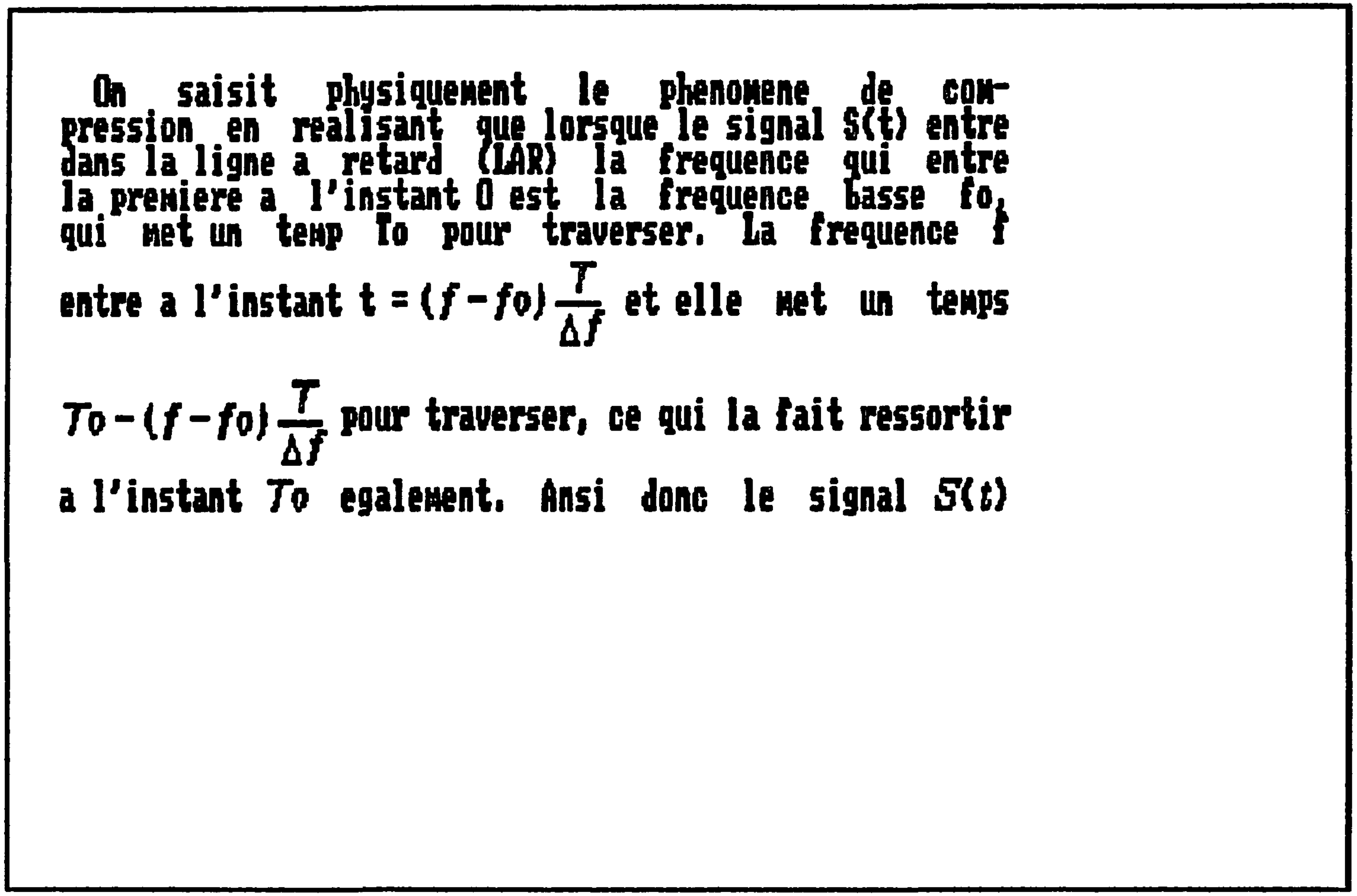
Un signal utile $S(t)$ traversant un tel filtre adapte donne a la sortie (a un retard To pres et un dephasage pres de la porteuse) un signal dont la transformee de Fourier est reelle, constante entre fo et $fo+\Delta f$, et nulle de part et d'autre de fo et de $fo+\Delta f$, c'est-a-dire un signal de frequence porteuse $fo+\Delta f/2$ et dont l'enveloppe a la forme indiquee a la figure 5, ou l'on a represente simultanement le signal S(t) et le signal $S(t)$ correspondant obtenu a la sortie du filtre adapte. On comprend le nom de recepteur a compression d'impulsion donne a ce genre de filtre adapte : la " largeur " (a 3 dB) du signal con-

Figure 12.13. Image doc5ra



prime etant egale a 1/Uf, le rapport de compression est de $\frac{T}{1/\Delta f} = T\Delta f$

Figure 12.14. Image doc5rb



On saisit physiquement le phenomene de compression en realisant que lorsque le signal S(t) entre dans la ligne a retard (LAR) la frequence qui entre la premiere a l'instant 0 est la frequence basse fo, qui met un temp To pour traverser. La frequence f entre a l'instant $t = (f - fo)\frac{T}{\Delta f}$ et elle met un temps $To - (f - fo)\frac{T}{\Delta f}$ pour traverser, ce qui la fait ressortir a l'instant To egalement. Ansi donc le signal S(t)

Figure 12.15. Image doc5rc

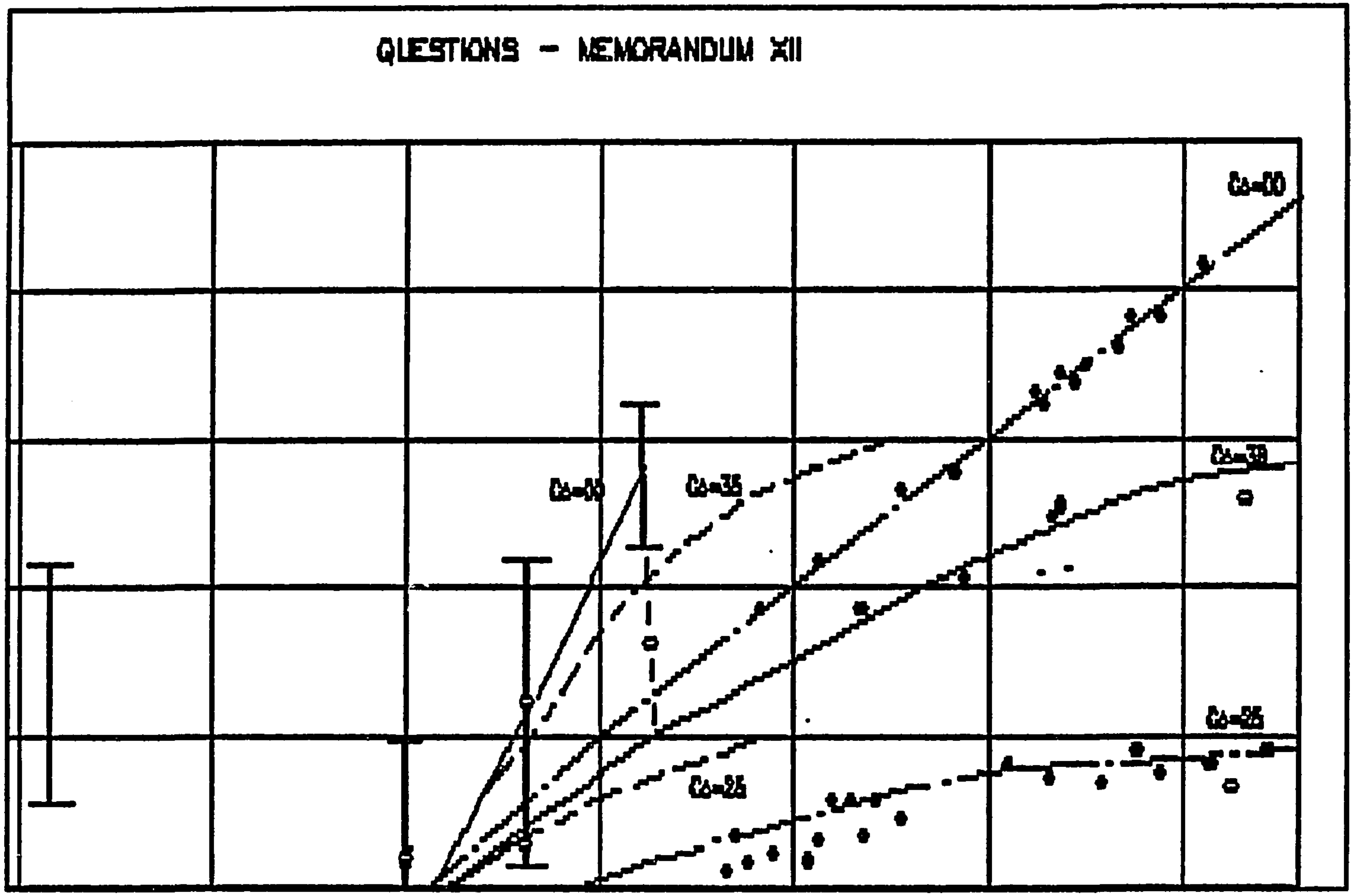


Figure 12.16. Image doc6a

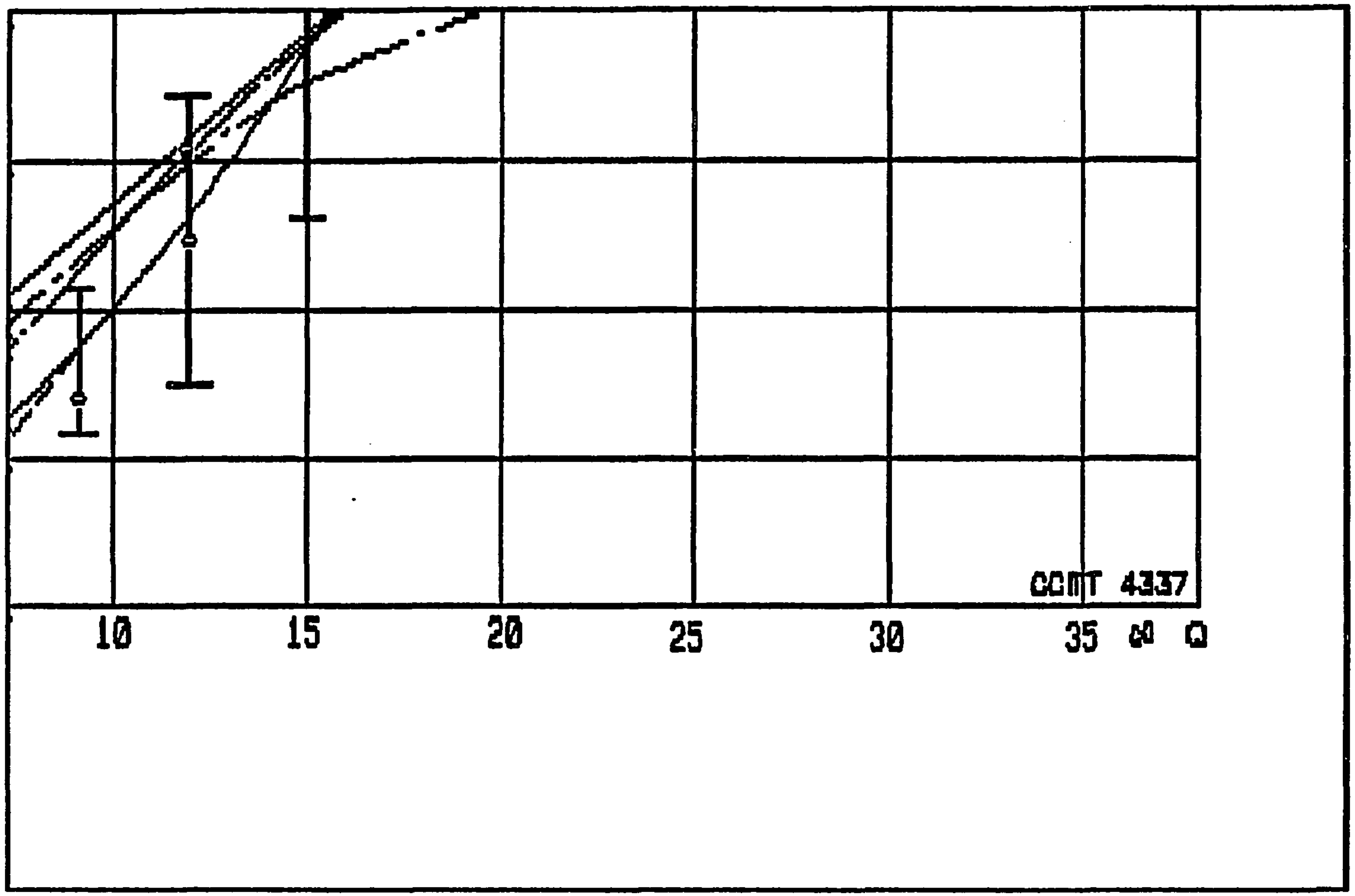
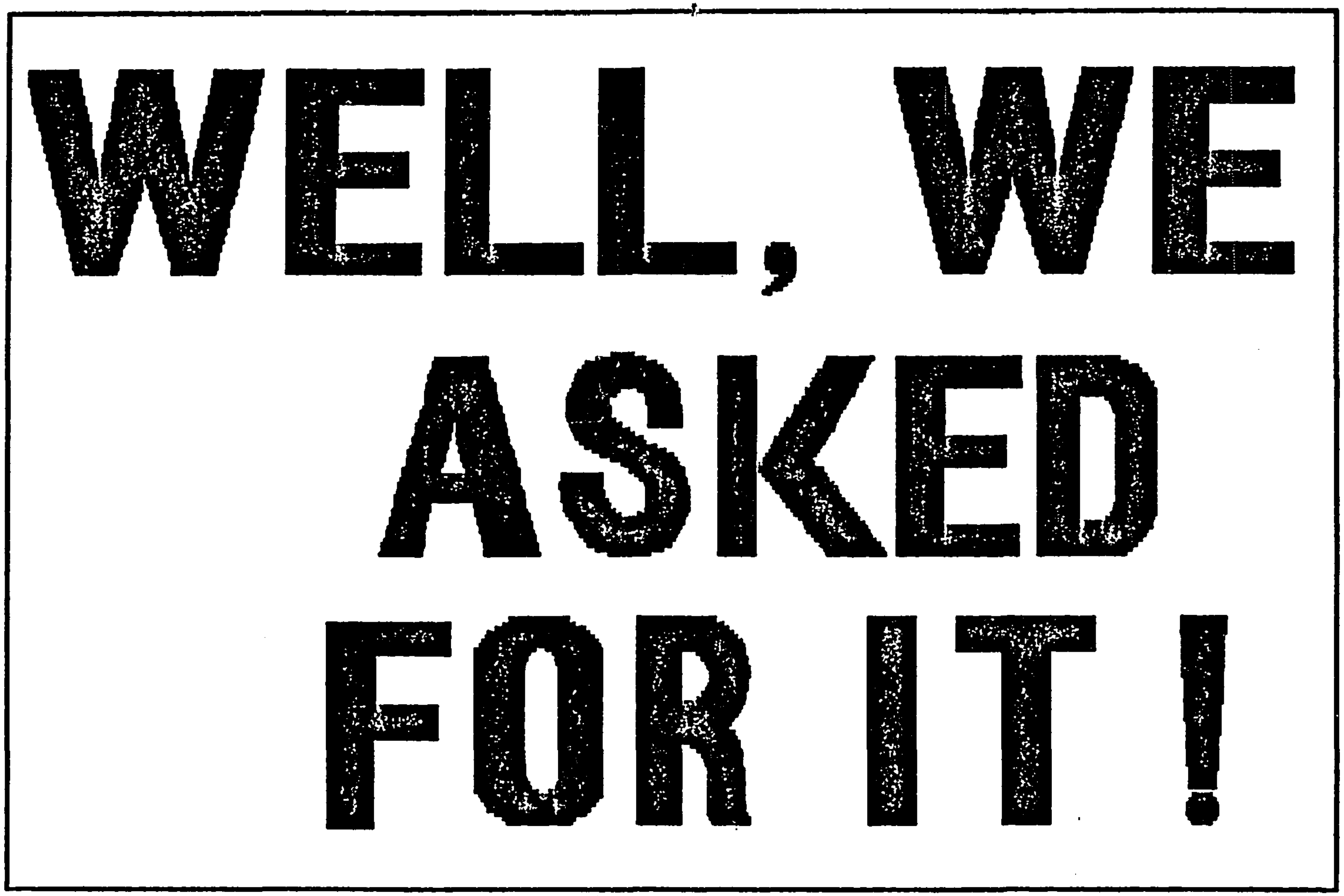


Figure 12.17. Image doc6b



WELL, WE
ASKED
FOR IT!

Figure 12.18. Image doc8

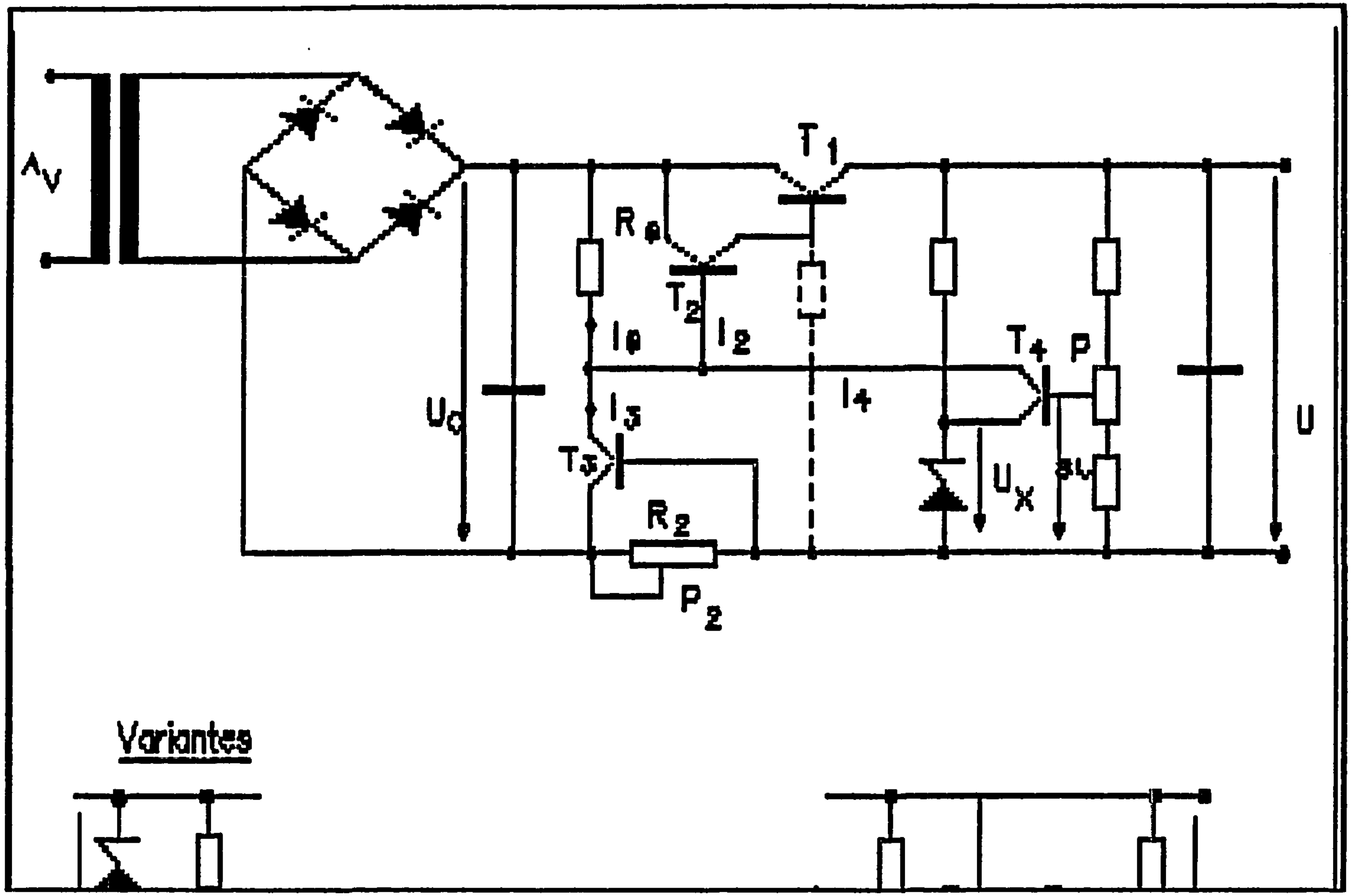


Figure 12.19. Image frnch3a

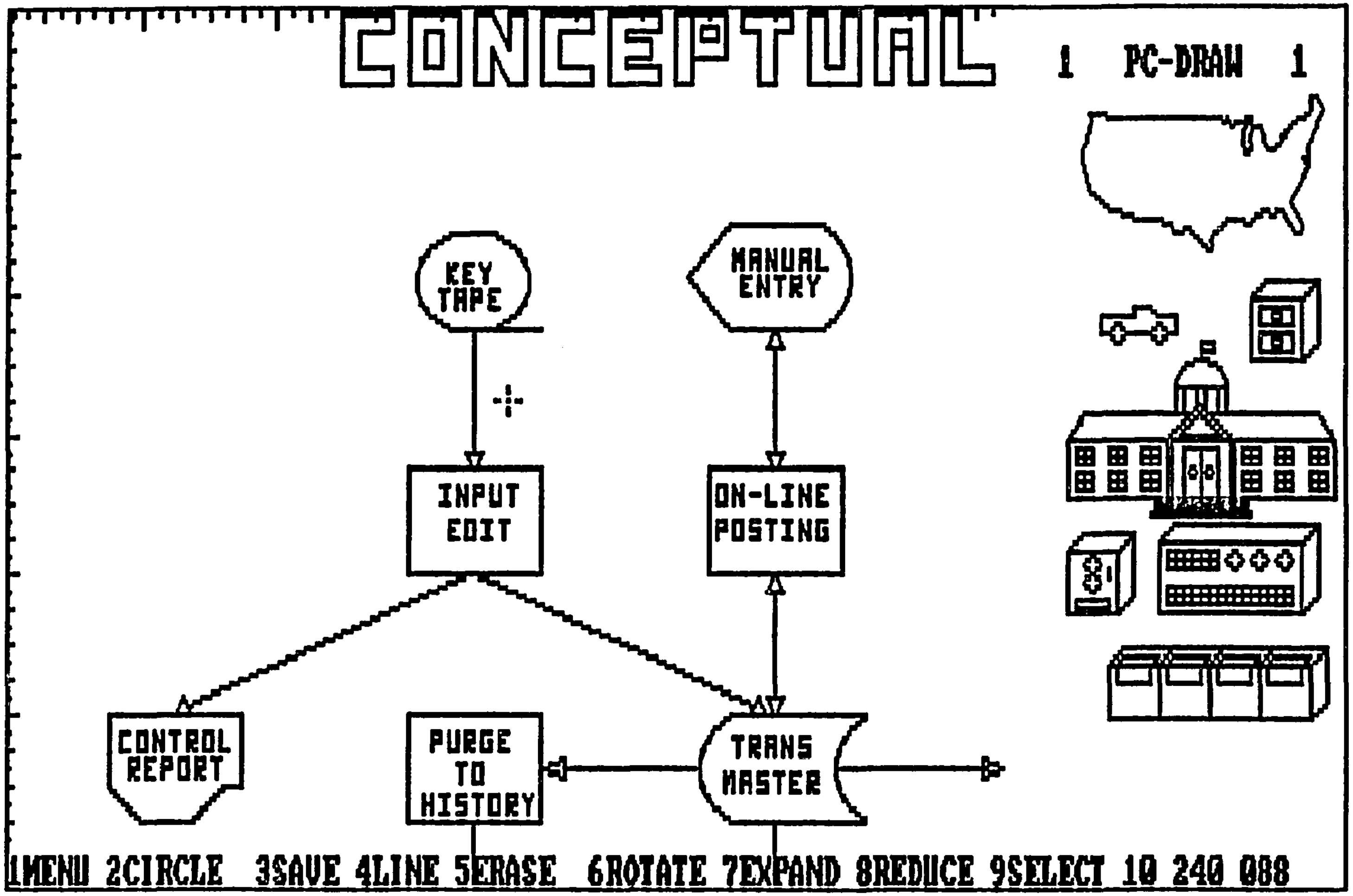


Figure 12.20. Image flowchrt

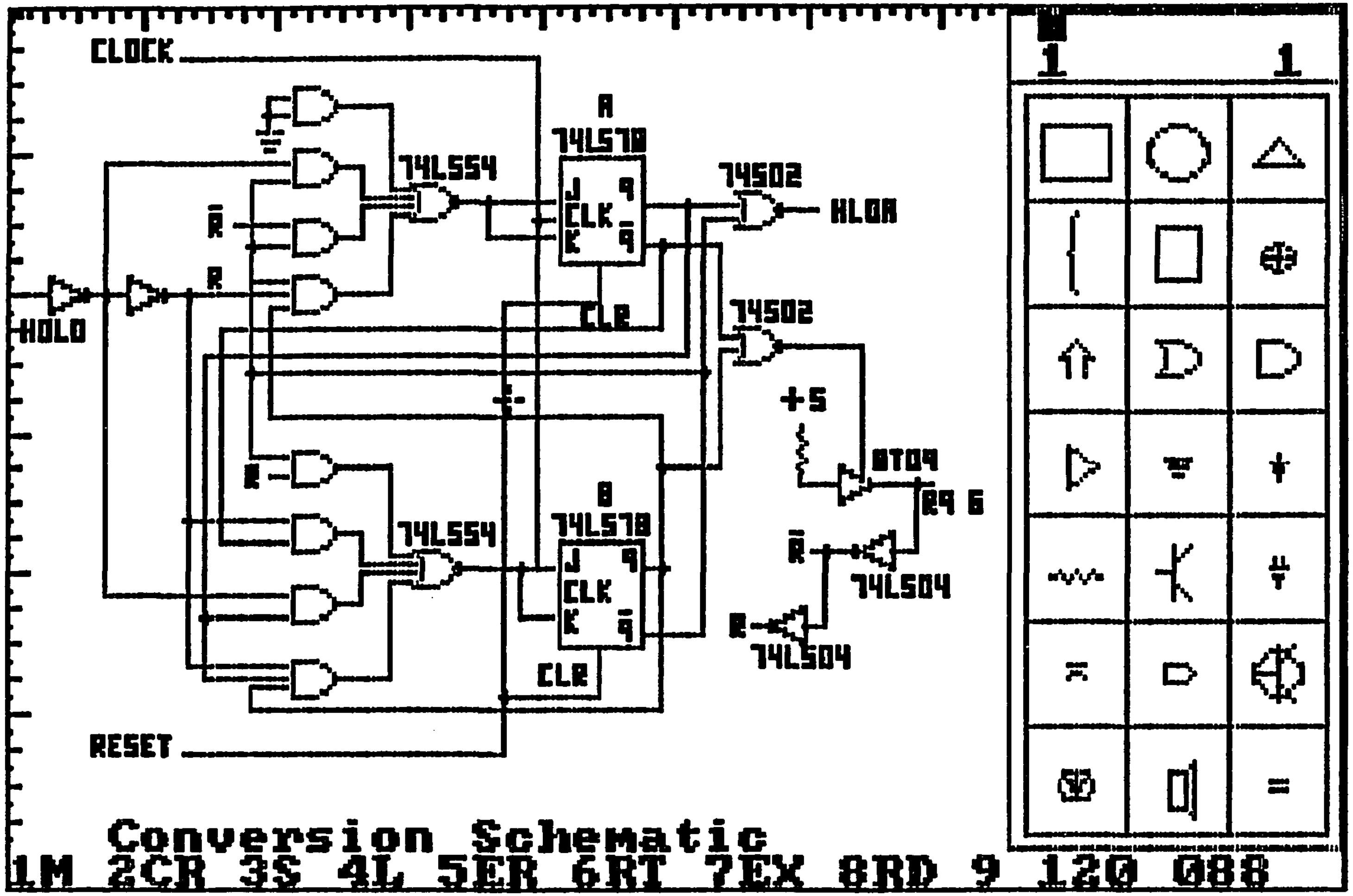
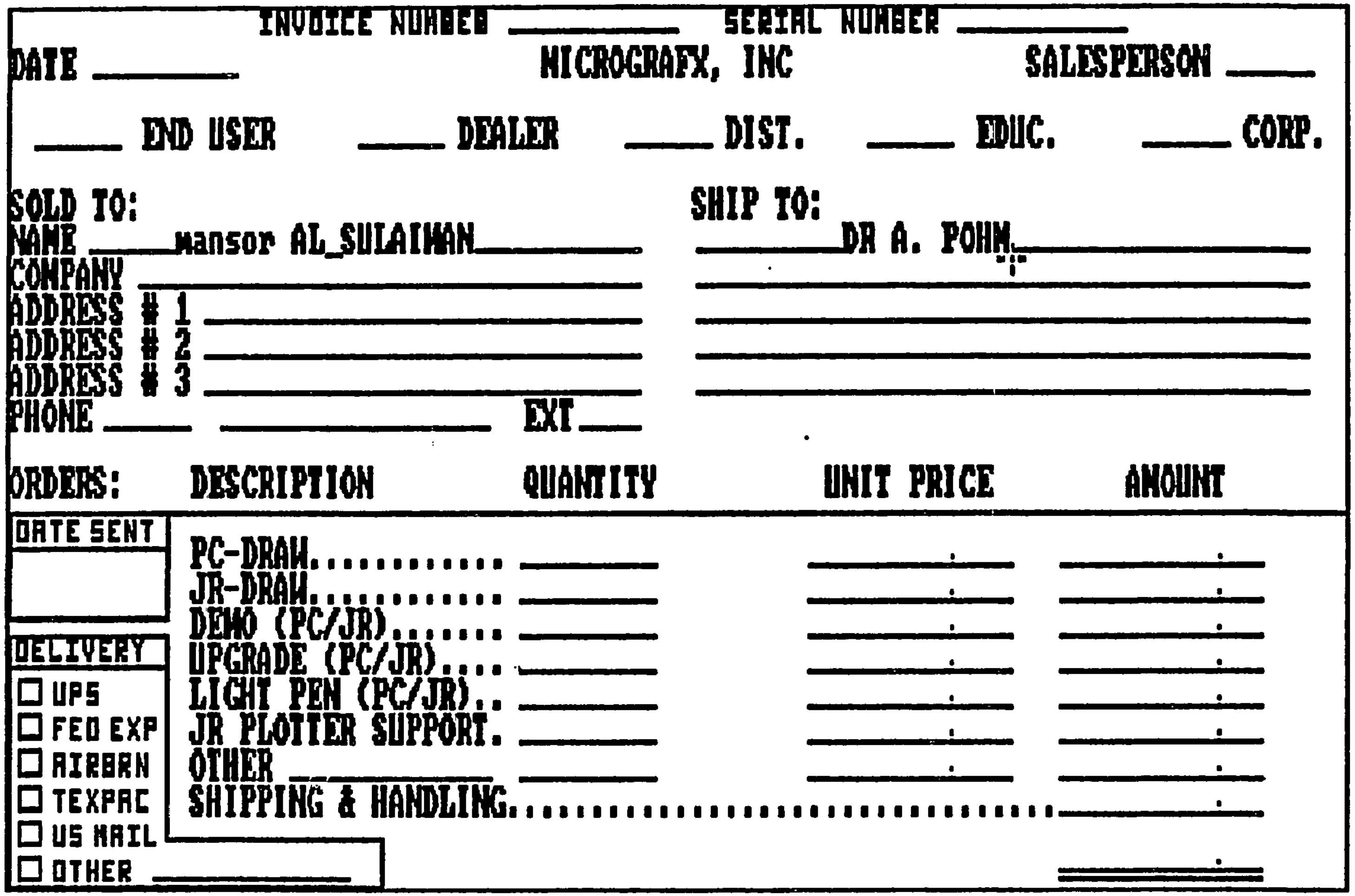


Figure 12.21. Image electrc



INVOICE NUMBER ________ SERIAL NUMBER ________

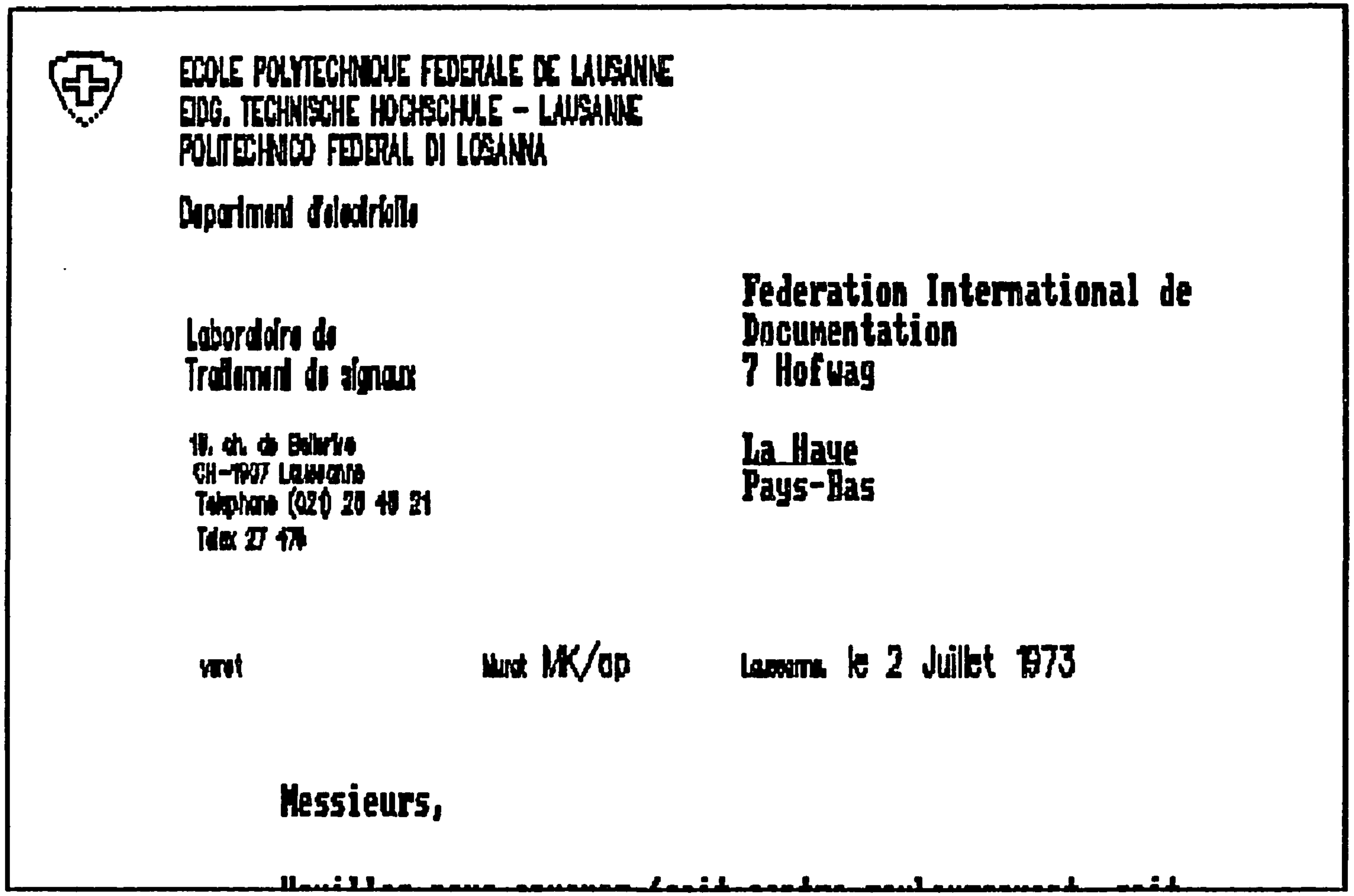
DATE ______ MICROGRAFX, INC SALESPERSON ____

____ END USER ____ DEALER ____ DIST. ____ EDUC. ____ CORP.

SOLD TO:	SHIP TO:
NAME ____mansor AL_SULAIMAN________	______DR A. POHM______
COMPANY ______________	______________
ADDRESS # 1 ______________	______________
ADDRESS # 2 ______________	______________
ADDRESS # 3 ______________	______________
PHONE ____ ________ EXT ___	

ORDERS:	DESCRIPTION	QUANTITY	UNIT PRICE	AMOUNT
DATE SENT	PC-DRAW...........	______	_____._____	_____._____
	JR-DRAW...........	______	_____._____	_____._____
	DEMO (PC/JR).......	______	_____._____	_____._____
DELIVERY	UPGRADE (PC/JR)....	______	_____._____	_____._____
☐ UPS	LIGHT PEN (PC/JR)..	______	_____._____	_____._____
☐ FED EXP	JR PLOTTER SUPPORT.	______	_____._____	_____._____
☐ AIRBRN	OTHER __________	______	_____._____	_____._____
☐ TEXPAC	SHIPPING & HANDLING.............................			_____._____
☐ US MAIL				
☐ OTHER ________				_____._____

Figure 12.22. Image ordrfrm



ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE
EIDG. TECHNISCHE HOCHSCHULE - LAUSANNE
POLITECHNICO FEDERAL DI LOSANNA

Departement d'electricite

Laboratoire de
Traitement de signaux

16, ch. de Bellerive
CH-1007 Lausanne
Telephone (021) 26 46 21
Telex 27 474

Federation International de
Documentation
7 Hofwag

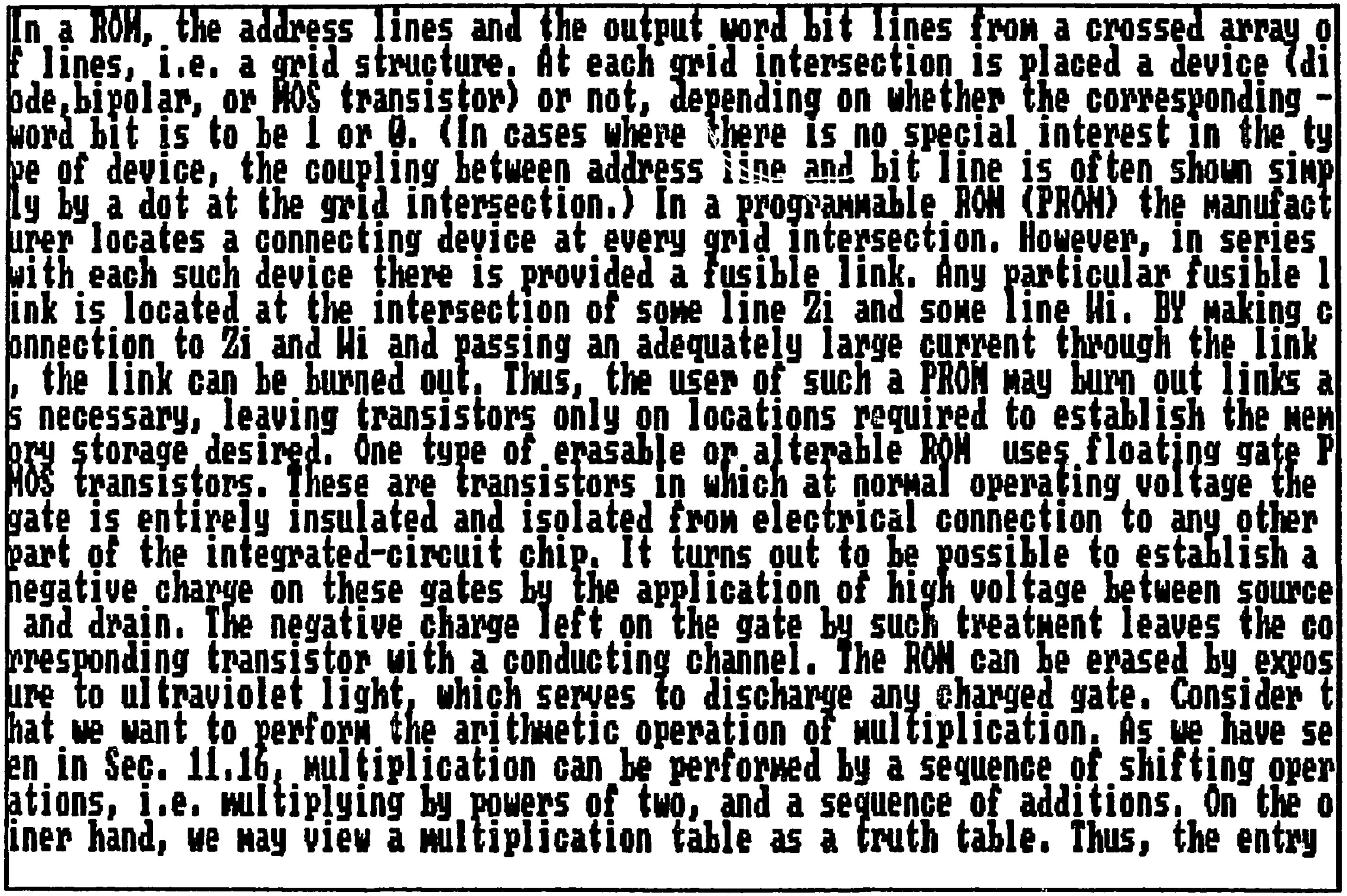
La Haye
Pays-Bas

MK/op

Lausanne le 2 Juillet 1973

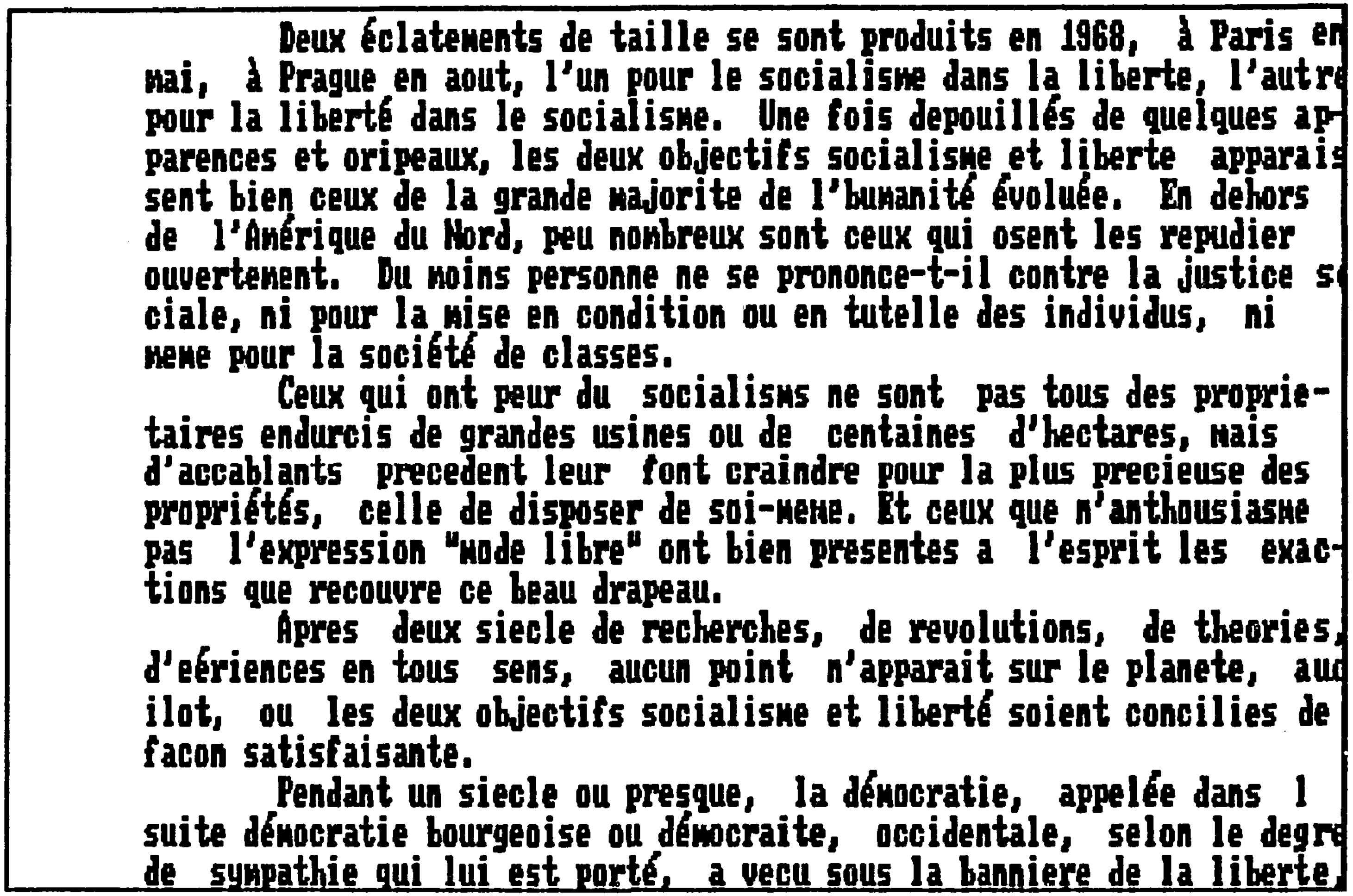
Messieurs,

Figure 12.23. Image frnchla



In a ROM, the address lines and the output word bit lines from a crossed array o
f lines, i.e. a grid structure. At each grid intersection is placed a device (di
ode,bipolar, or MOS transistor) or not, depending on whether the corresponding -
word bit is to be 1 or 0. (In cases where there is no special interest in the ty
pe of device, the coupling between address line and bit line is often shown simp
ly by a dot at the grid intersection.) In a programmable ROM (PROM) the manufact
urer locates a connecting device at every grid intersection. However, in series
with each such device there is provided a fusible link. Any particular fusible l
ink is located at the intersection of some line Zi and some line Wi. BY making c
onnection to Zi and Wi and passing an adequately large current through the link
, the link can be burned out. Thus, the user of such a PROM may burn out links a
s necessary, leaving transistors only on locations required to establish the mem
ory storage desired. One type of erasable or alterable ROM uses floating gate P
MOS transistors. These are transistors in which at normal operating voltage the
gate is entirely insulated and isolated from electrical connection to any other
part of the integrated-circuit chip. It turns out to be possible to establish a
negative charge on these gates by the application of high voltage between source
and drain. The negative charge left on the gate by such treatment leaves the co
rresponding transistor with a conducting channel. The ROM can be erased by expos
ure to ultraviolet light, which serves to discharge any charged gate. Consider t
hat we want to perform the arithmetic operation of multiplication. As we have se
en in Sec. 11.16, multiplication can be performed by a sequence of shifting oper
ations, i.e. multiplying by powers of two, and a sequence of additions. On the o
iner hand, we may view a multiplication table as a truth table. Thus, the entry

Figure 12.24. Image romtxt



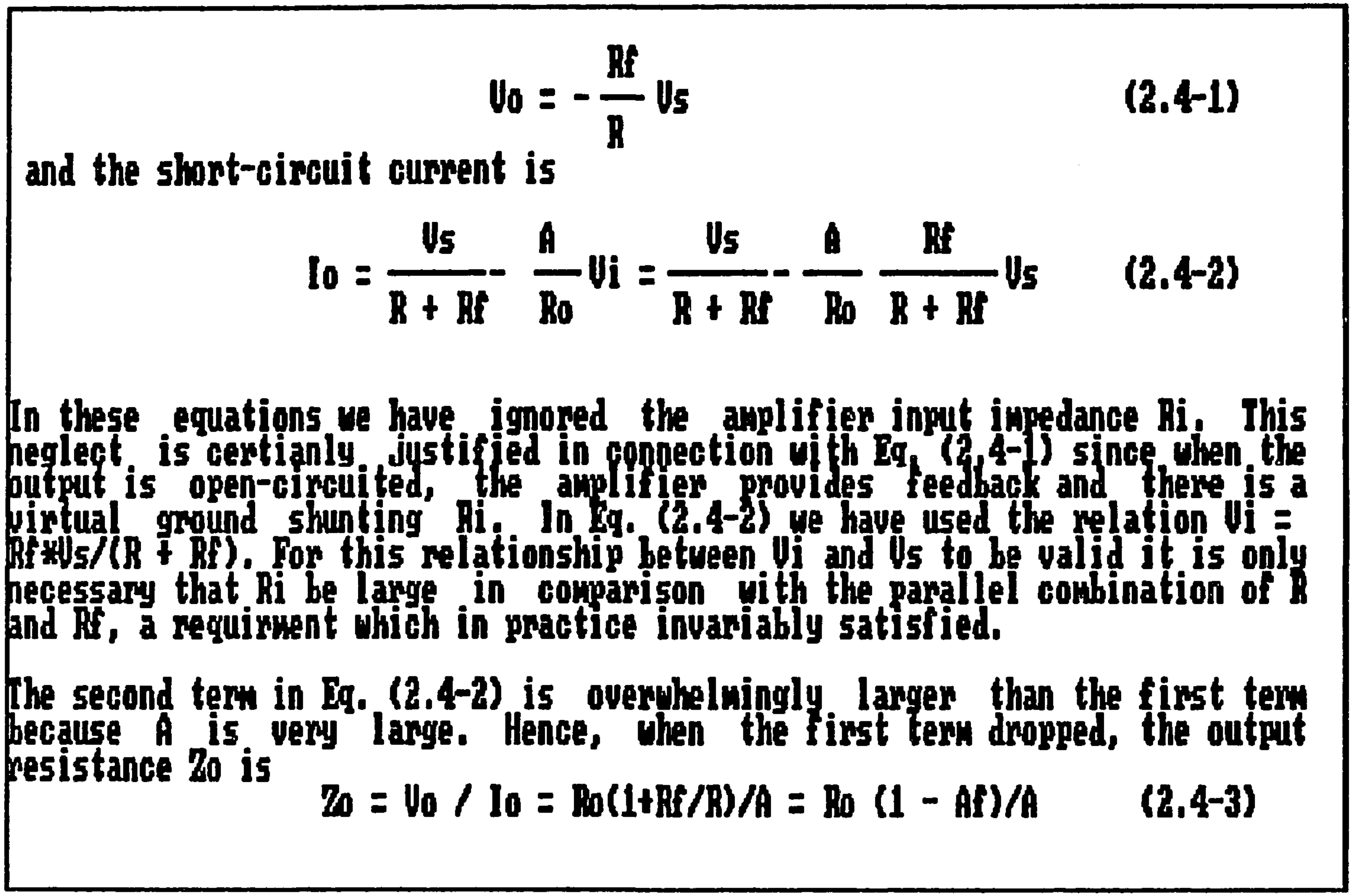
Deux éclatements de taille se sont produits en 1968, à Paris e
mai, à Prague en aout, l'un pour le socialisme dans la liberte, l'autr
pour la liberté dans le socialisme. Une fois depouillés de quelques ap-
parences et oripeaux, les deux objectifs socialisme et liberte apparai
sent bien ceux de la grande majorite de l'bumanité évoluée. En dehors
de l'Amérique du Nord, peu nombreux sont ceux qui osent les repudier
ouvertement. Du moins personne ne se prononce-t-il contre la justice s
ciale, ni pour la mise en condition ou en tutelle des individus, ni
meme pour la société de classes.

Ceux qui ont peur du socialisms ne sont pas tous des proprie-
taires endurcis de grandes usines ou de centaines d'hectares, mais
d'accablants precedent leur font craindre pour la plus precieuse des
propriétés, celle de disposer de soi-meme. Et ceux que n'anthousiasme
pas l'expression "mode libre" ont bien presentes a l'esprit les exac-
tions que recouvre ce beau drapeau.

Apres deux siecle de recherches, de revolutions, de theories,
d'eériences en tous sens, aucun point n'apparait sur le planete, au
ilot, ou les deux objectifs socialisme et liberté soient concilies de
facon satisfaisante.

Pendant un siecle ou presque, la démocratie, appelée dans l
suite démocratie bourgeoise ou démocraite, occidentale, selon le degr
de sympathie qui lui est porté, a vecu sous la banniere de la liberte,

Figure 12.25. Image frnch2a



$$Vo = -\frac{Rf}{R} Vs \qquad (2.4\text{-}1)$$

and the short-circuit current is

$$Io = \frac{Vs}{R + Rf} - \frac{A}{Ro} Vi = \frac{Vs}{R + Rf} - \frac{A}{Ro} \frac{Rf}{R + Rf} Vs \qquad (2.4\text{-}2)$$

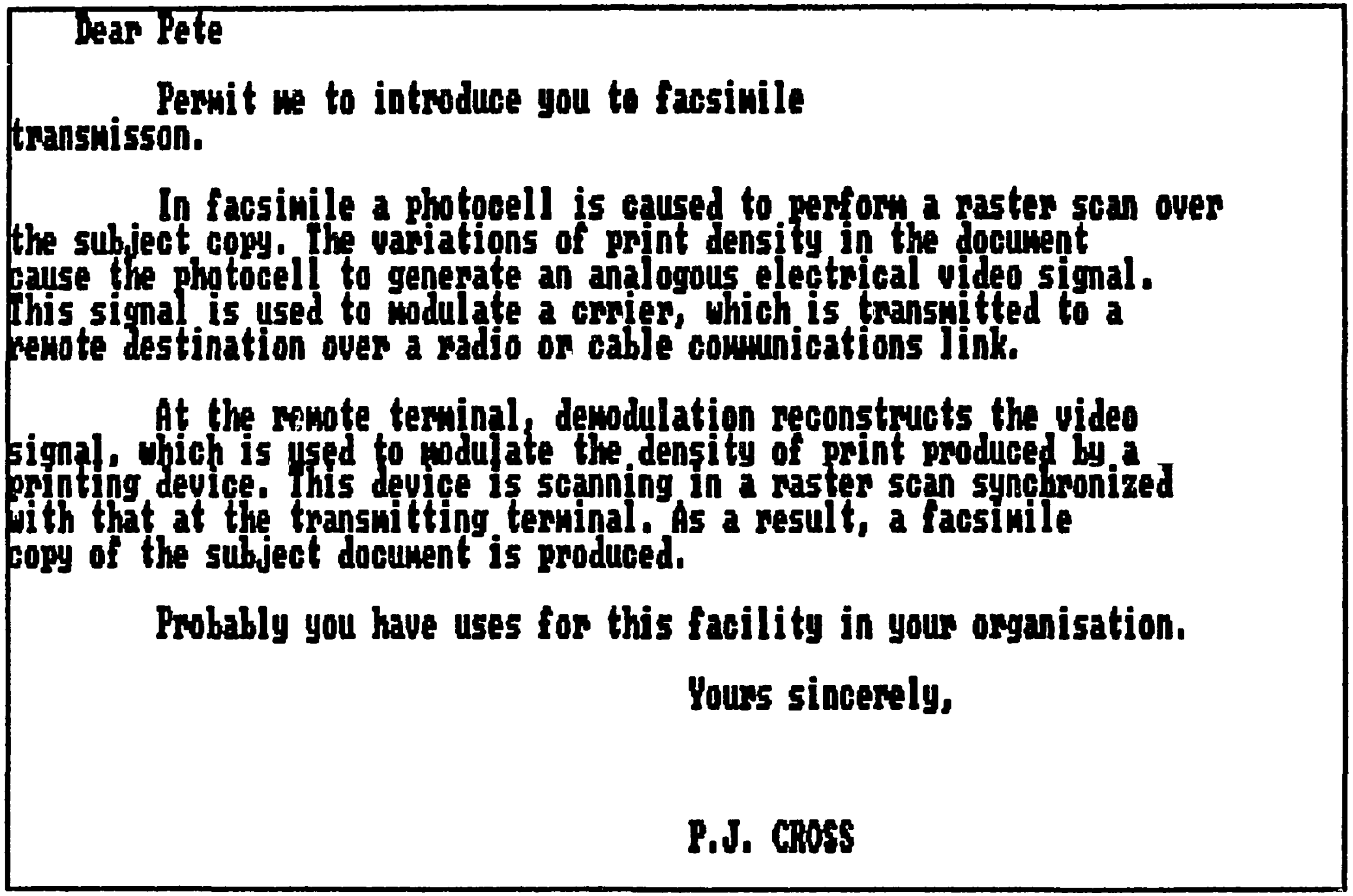
In these equations we have ignored the amplifier input impedance Ri. This neglect is certianly justified in connection with Eq. (2.4-1) since when the output is open-circuited, the amplifier provides feedback and there is a virtual ground shunting Ri. In Eq. (2.4-2) we have used the relation Vi = Rf*Vs/(R + Rf). For this relationship between Vi and Vs to be valid it is only necessary that Ri be large in comparison with the parallel combination of R and Rf, a requirment which in practice invariably satisfied.

The second term in Eq. (2.4-2) is overwhelmingly larger than the first term because A is very large. Hence, when the first term dropped, the output resistance Zo is

$$Zo = Vo / Io = Ro(1+Rf/R)/A = Ro\,(1 - Af)/A \qquad (2.4\text{-}3)$$

Figure 12.26. Image page1

]



Dear Pete

Permit me to introduce you to facsimile transmisson.

In facsimile a photocell is caused to perform a raster scan over the subject copy. The variations of print density in the document cause the photocell to generate an analogous electrical video signal. This signal is used to modulate a crrier, which is transmitted to a remote destination over a radio or cable communications link.

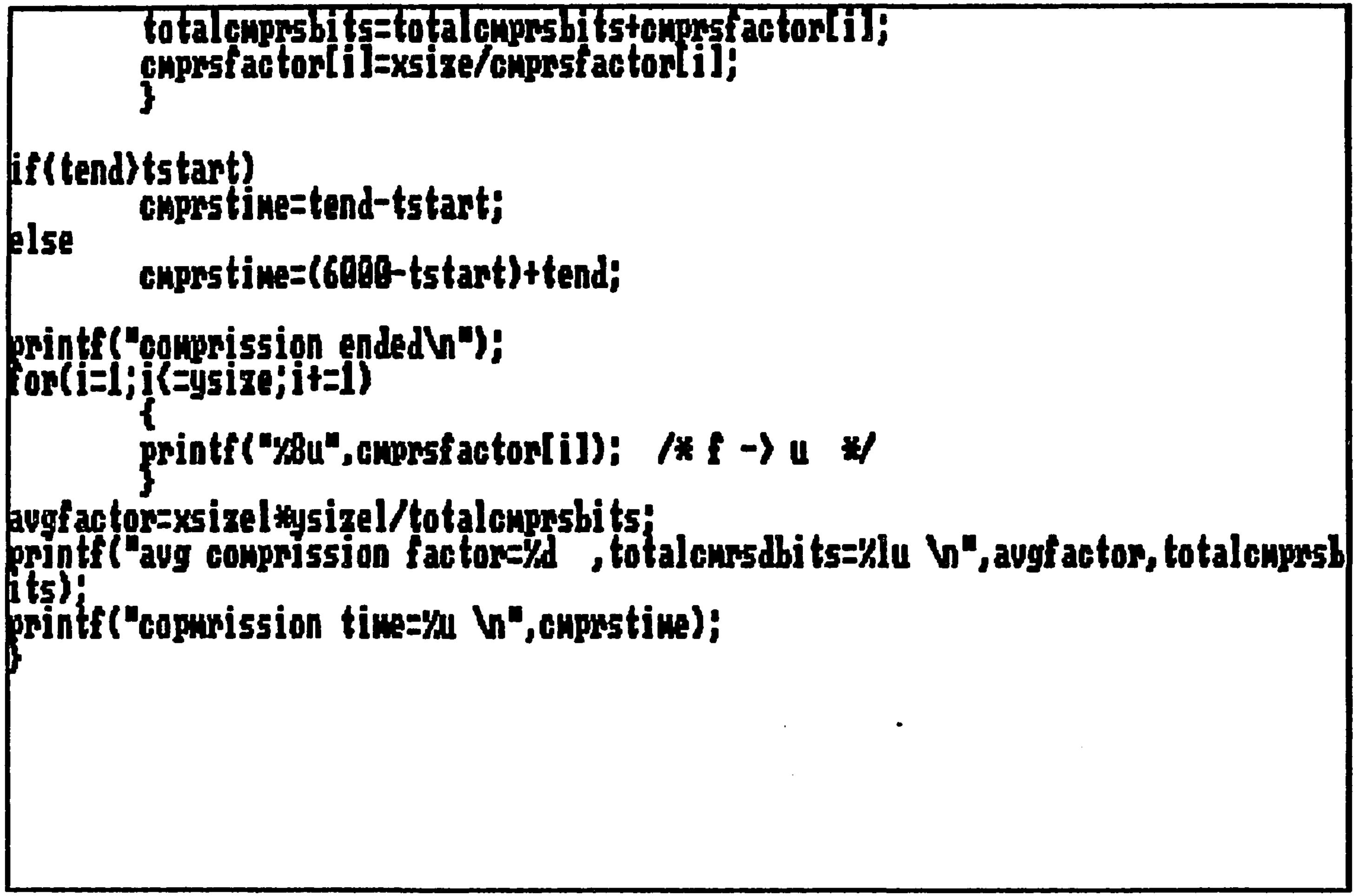
At the remote terminal, demodulation reconstructs the video signal, which is used to modulate the density of print produced by a printing device. This device is scanning in a raster scan synchronized with that at the transmitting terminal. As a result, a facsimile copy of the subject document is produced.

Probably you have uses for this facility in your organisation.

Yours sincerely,

P.J. CROSS

Figure 12.27. Image docl-2

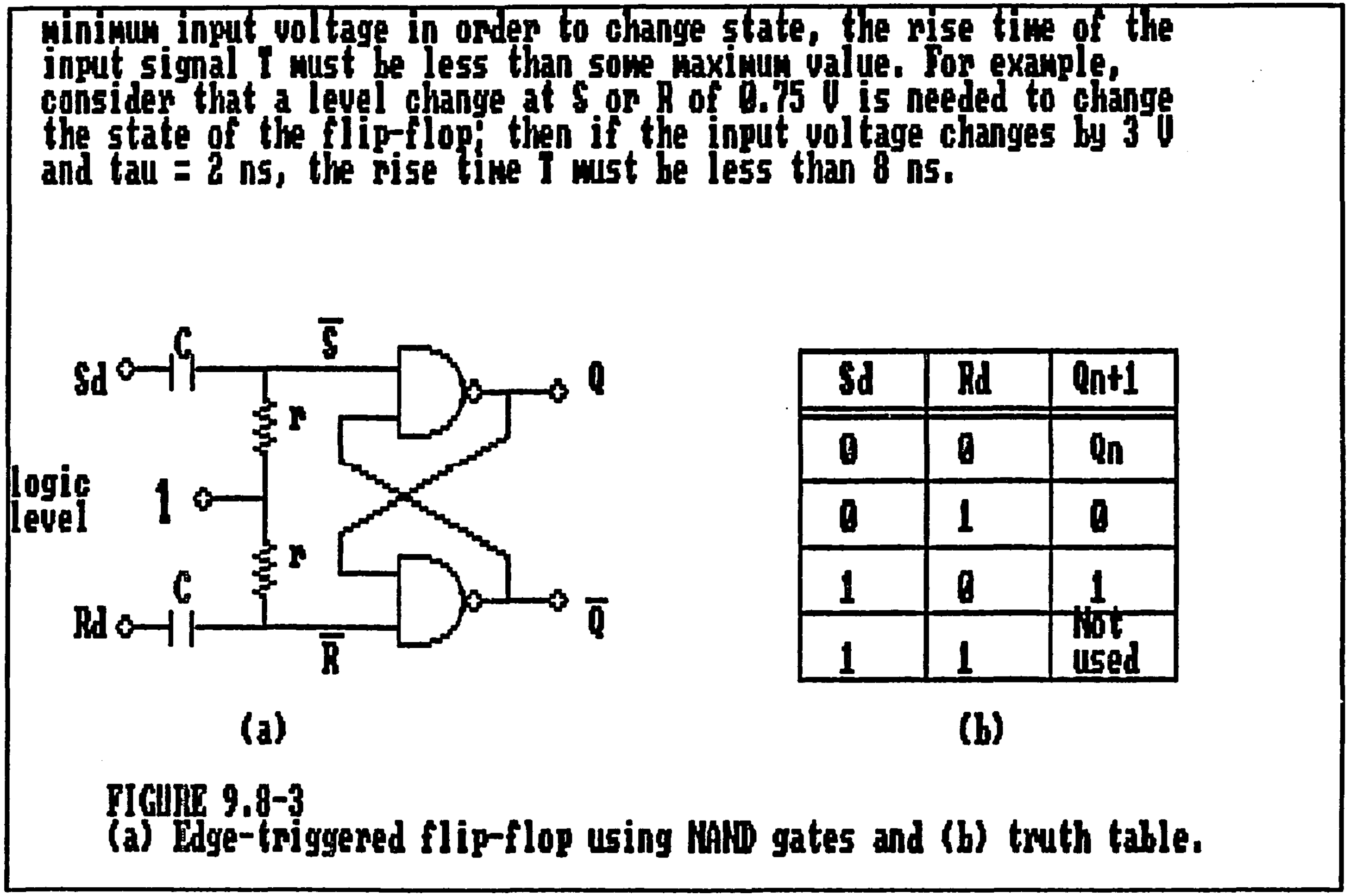


```
        totalcmprsbits=totalcmprsbits+cmprsfactor[i];
        cmprsfactor[i]=xsize/cmprsfactor[i];
        }

if(tend>tstart)
        cmprstime=tend-tstart;
else
        cmprstime=(6000-tstart)+tend;

printf("comprission ended\n");
for(i=1;i<=ysize;i+=1)
        {
        printf("%8u",cmprsfactor[i]);  /* f -> u  */
        }
avgfactor=xsize1*ysize1/totalcmprsbits;
printf("avg comprission factor=%d  ,totalcmrsdbits=%lu \n",avgfactor,totalcmprsb
its);
printf("copmrission time=%u \n",cmprstime);
}
```

Figure 12.28. Image cprog



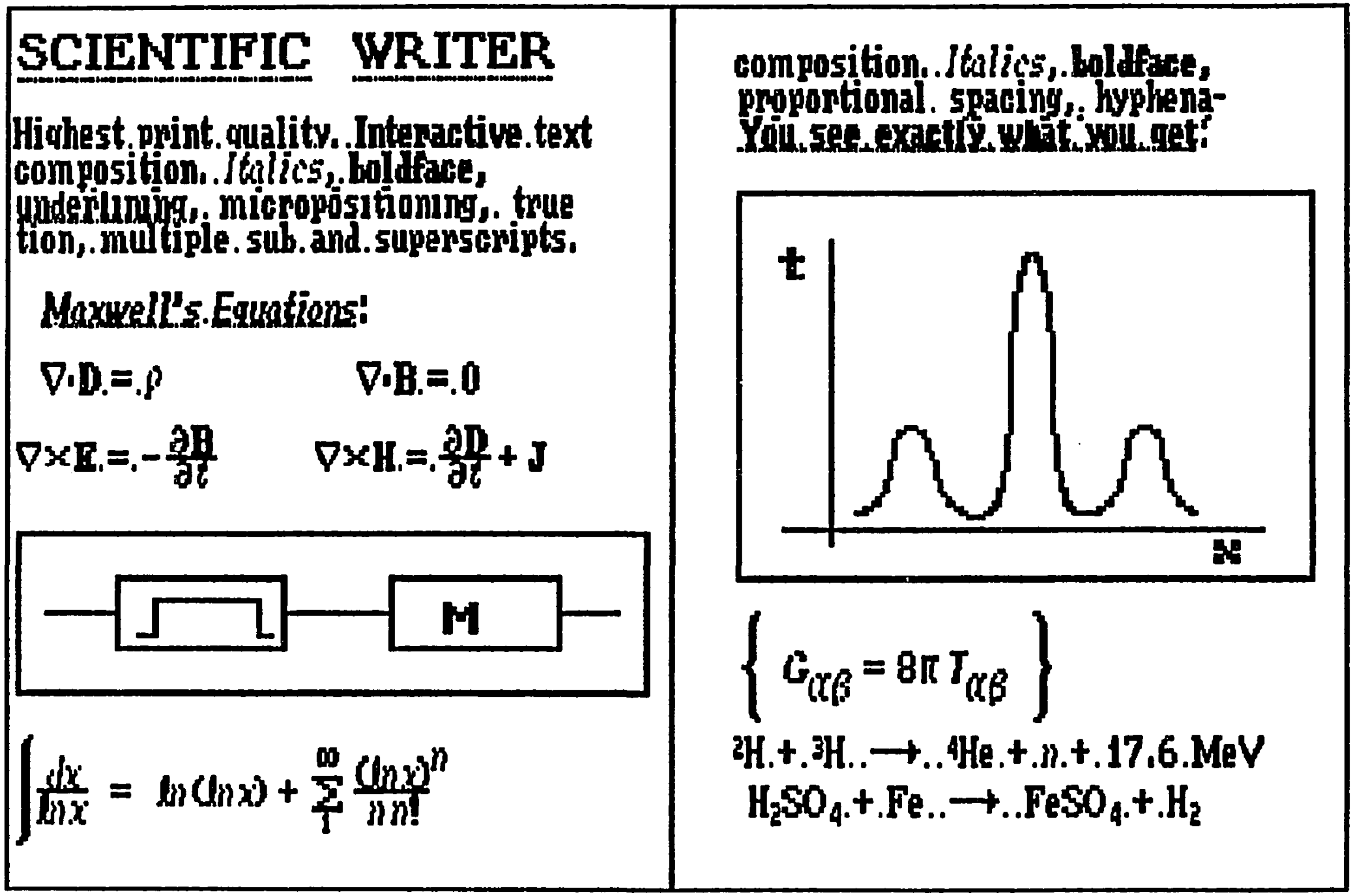
minimum input voltage in order to change state, the rise time of the input signal T must be less than some maximum value. For example, consider that a level change at S or R of 0.75 V is needed to change the state of the flip-flop; then if the input voltage changes by 3 V and tau = 2 ns, the rise time T must be less than 8 ns.

Sd	Rd	Qn+1
0	0	Qn
0	1	0
1	0	1
1	1	Not used

(b)

FIGURE 9.8-3
(a) Edge-triggered flip-flop using NAND gates and (b) truth table.

Figure 12.29. Image pdraw3



SCIENTIFIC WRITER

Highest.print.quality..Interactive.text composition..*Italics*,.**boldface**, underlining,. micropositioning,. true tion,.multiple.sub.and.superscripts.

Maxwell's.Equations:

$\nabla\cdot D = \rho \qquad \nabla\cdot B = 0$

$\nabla\times E = -\frac{\partial B}{\partial t} \qquad \nabla\times H = \frac{\partial D}{\partial t} + J$

$\int \frac{dx}{\ln x} = \ln(\ln x) + \sum_{1}^{\infty} \frac{(\ln x)^n}{n\,n!}$

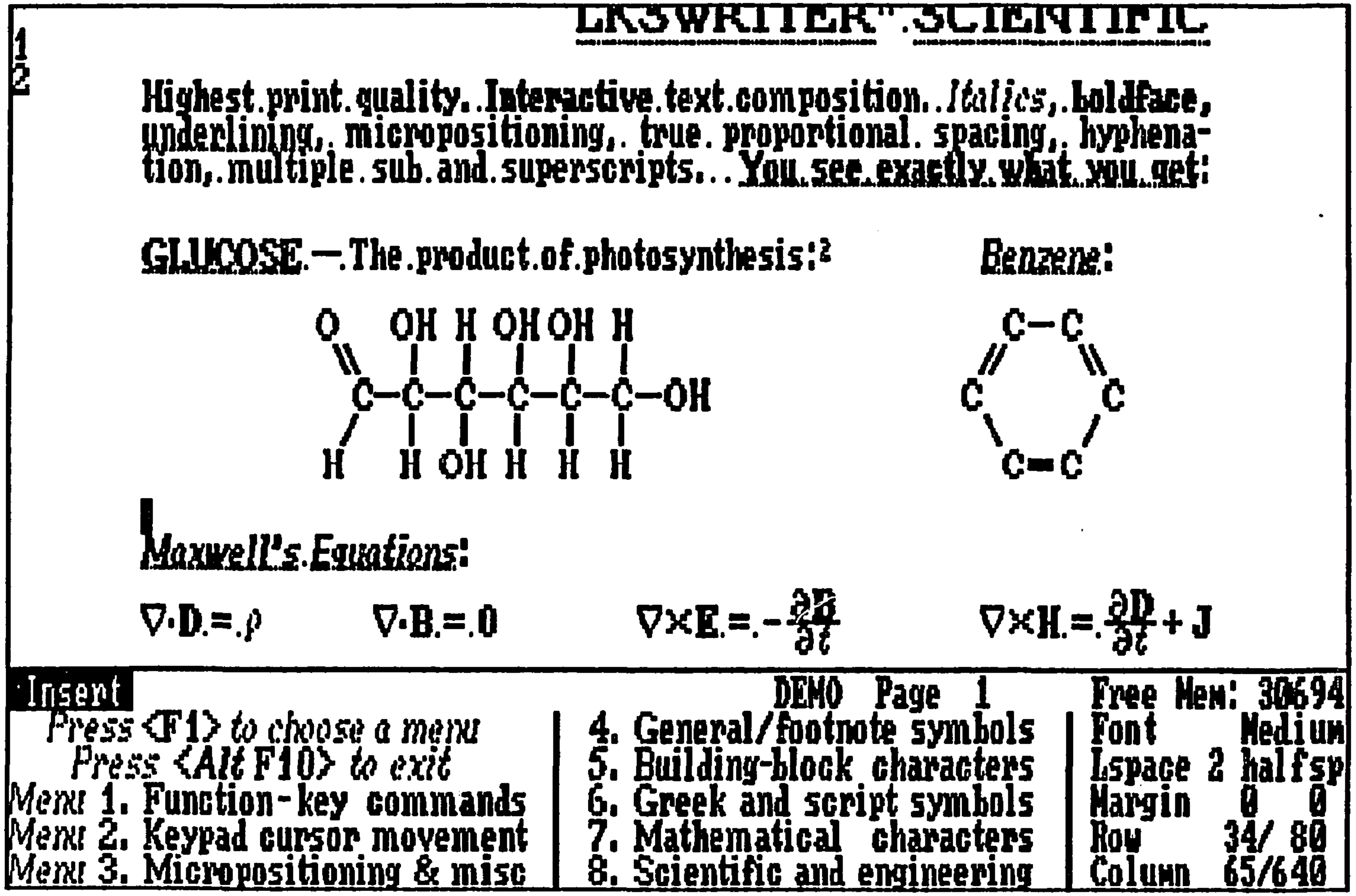
composition..*Italics*,.**boldface**, proportional. spacing,. hyphena- You.see.exactly.what.you.get!

$\left\{ G_{\alpha\beta} = 8\pi T_{\alpha\beta} \right\}$

$^2H + {}^3H \rightarrow {}^4He + n + 17.6\,MeV$

$H_2SO_4 + Fe \rightarrow FeSO_4 + H_2$

Figure 12.30. Image sciencel



LRSWRITER™.SCIENTIFIC

1
2

Highest.print.quality..Interactive.text.composition..*Italics*,.boldface, underlining,.micropositioning,.true.proportional.spacing,.hyphenation,.multiple.sub.and.superscripts...<u>You.see.exactly.what.you.get</u>!

<u>GLUCOSE</u>.—.The.product.of.photosynthesis:^{2}

***<u>Benzene</u>*:**

***<u>Maxwell's.Equations</u>*:**

$\nabla\cdot\mathbf{D} = \rho$ $\quad \nabla\cdot\mathbf{B} = 0$ $\quad \nabla\times\mathbf{E} = -\frac{\partial\mathbf{B}}{\partial t}$ $\quad \nabla\times\mathbf{H} = \frac{\partial\mathbf{D}}{\partial t} + \mathbf{J}$

```
Insert                                  DEMO  Page  1      Free Mem: 30694
  Press <F1> to choose a menu        | 4. General/footnote symbols | Font      Medium
    Press <Alt F10> to exit          | 5. Building-block characters | Lspace 2 halfsp
Menu 1. Function-key commands        | 6. Greek and script symbols  | Margin    0   0
Menu 2. Keypad cursor movement       | 7. Mathematical  characters  | Row      34/ 80
Menu 3. Micropositioning & misc      | 8. Scientific and engineering| Column  65/640
```

Figure 12.31. Image science2

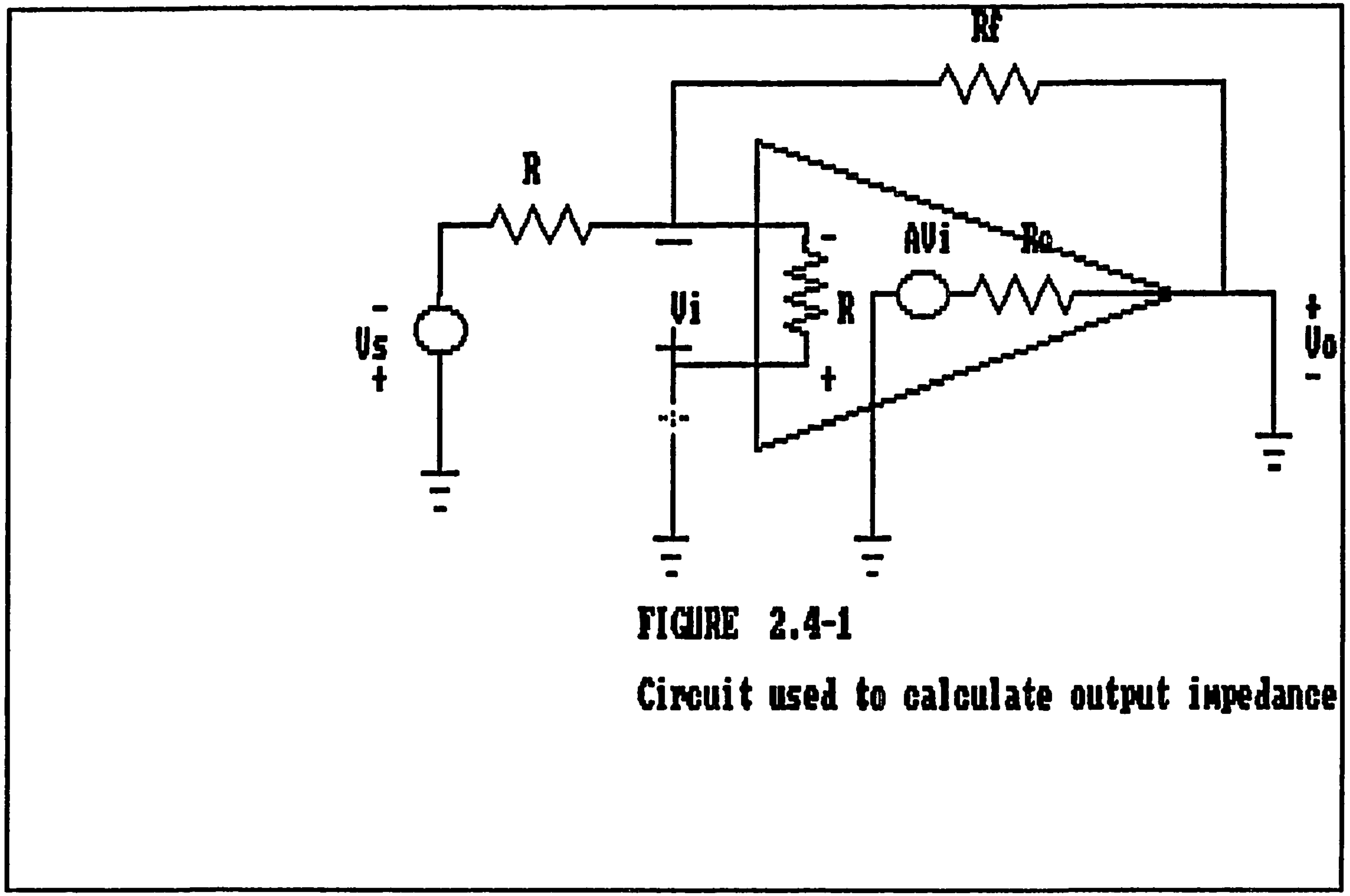


Figure 12.32. Image opampl

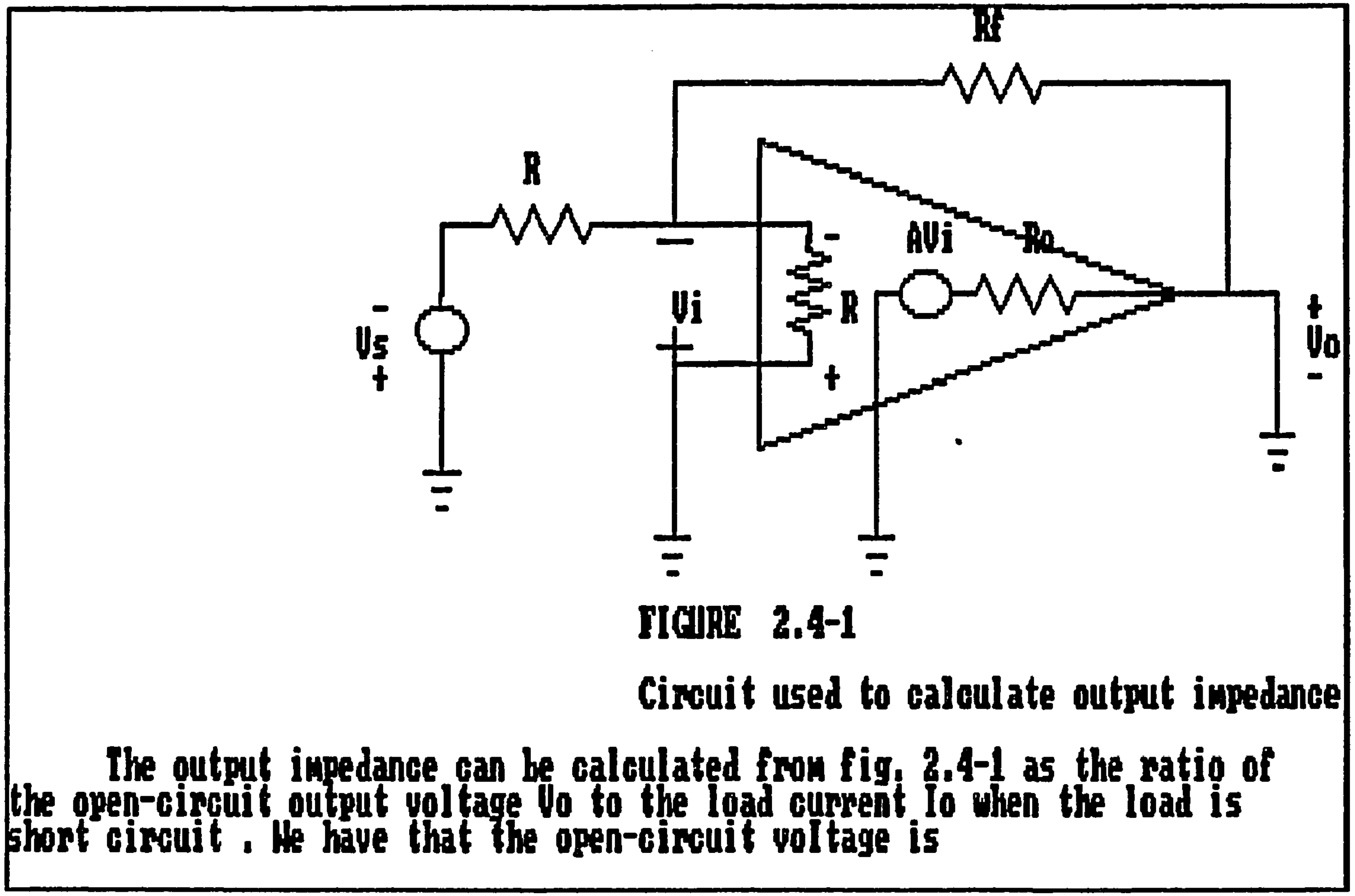


FIGURE 2.4-1

Circuit used to calculate output impedance

The output impedance can be calculated from fig. 2.4-1 as the ratio of the open-circuit output voltage Vo to the load current Io when the load is short circuit . We have that the open-circuit voltage is

Figure 12.33. Image opamp2

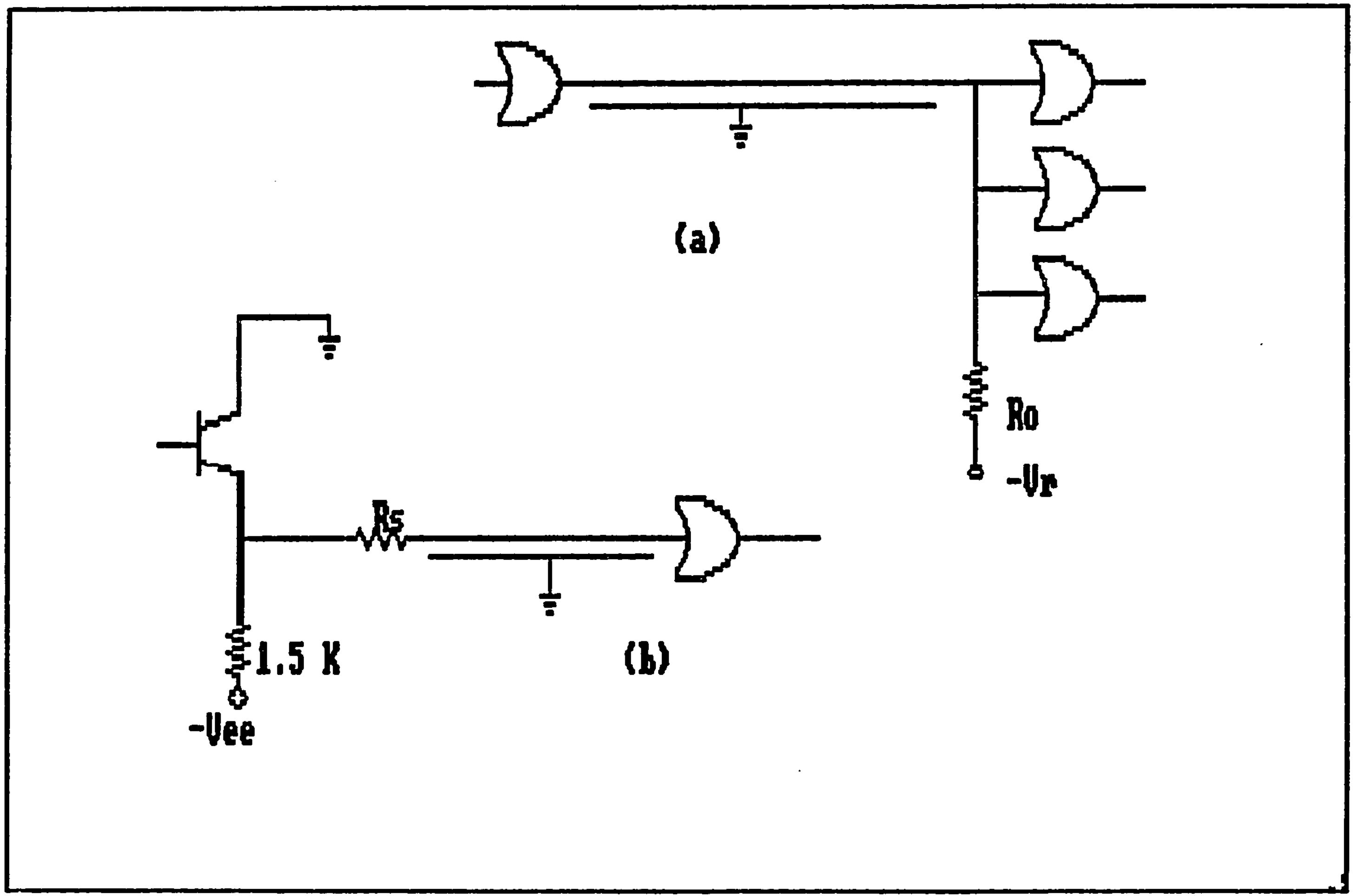


Figure 12.34. Image ecℓl

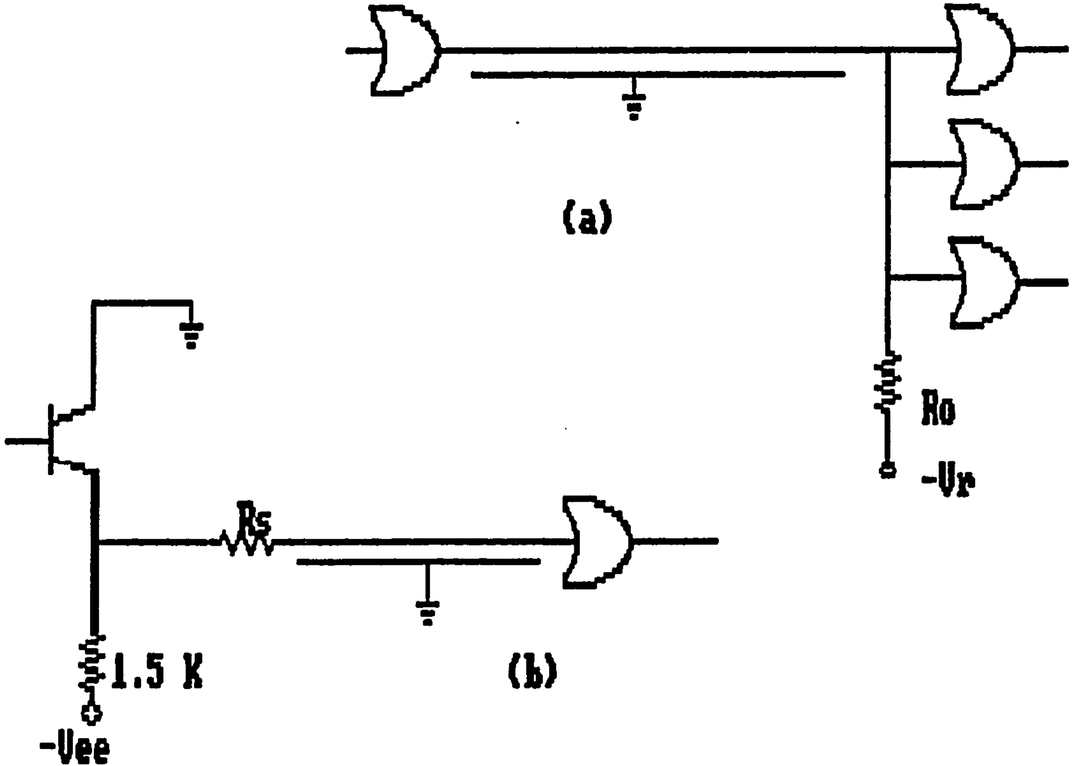


Figure 7.13-3

(a) The driven gates lumped at the end of a matched line; (b) a line matched

Figure 12.35. Image ecℓ2

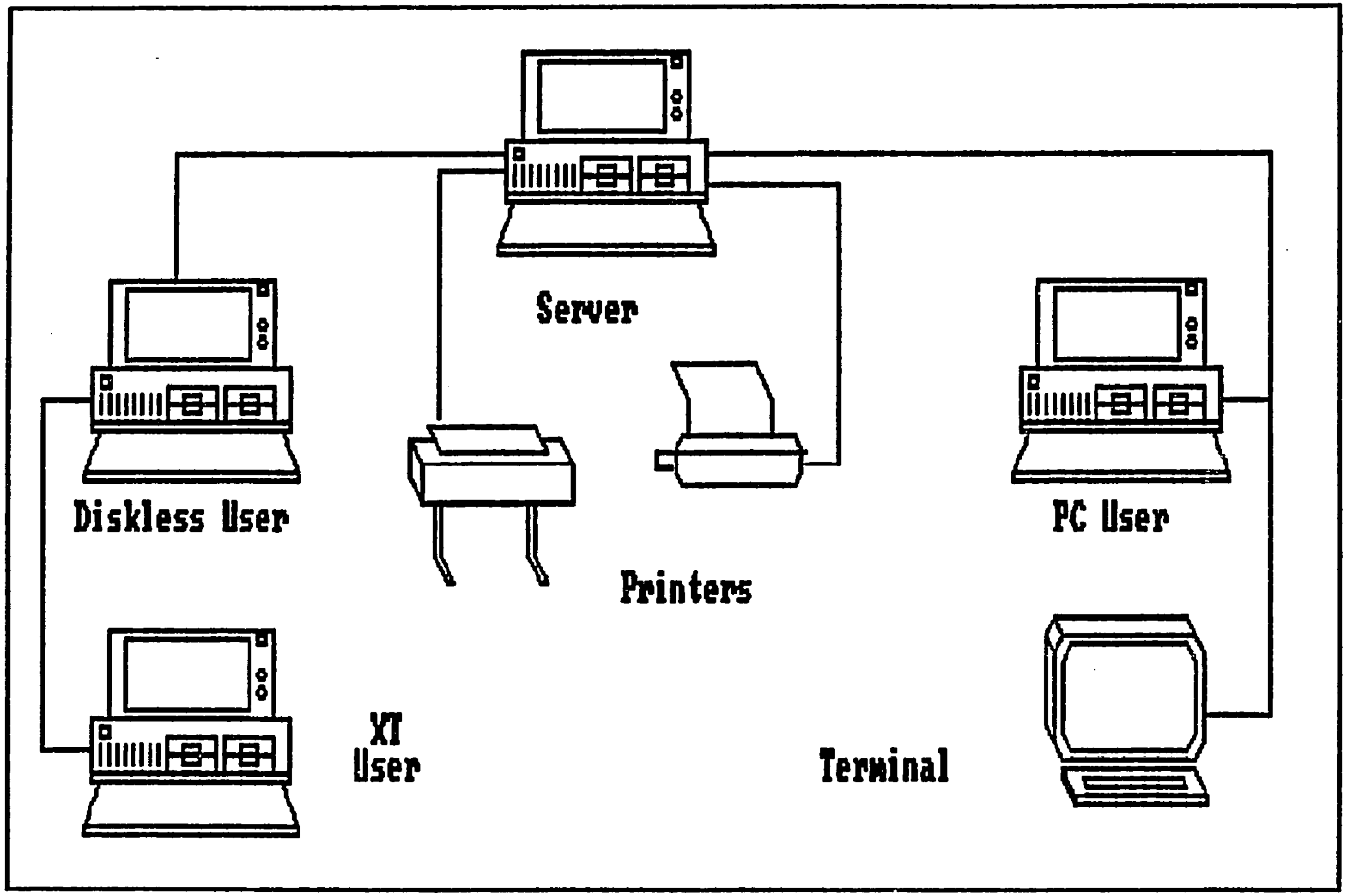


Figure 12.36. Image netwrk

DOCUMENT	RESYNC PERIOD CODED BIT		LOST RUNS (PELS)		LOST PELS (PELS)		DISPLACMENT PERCENTAGE 5 PELS
	AVREAGE	MEDIAN	AVREAGE	MEDIAN	AVREAGE	MEDIAN	
1	26	18	391	54	215	21	28%
4	24	16	122	29	77	13	39%
5	24	17	217	54	133	22	29%
7	27	17	140	69	69	33	29%

Figure 12.37. Image tablel

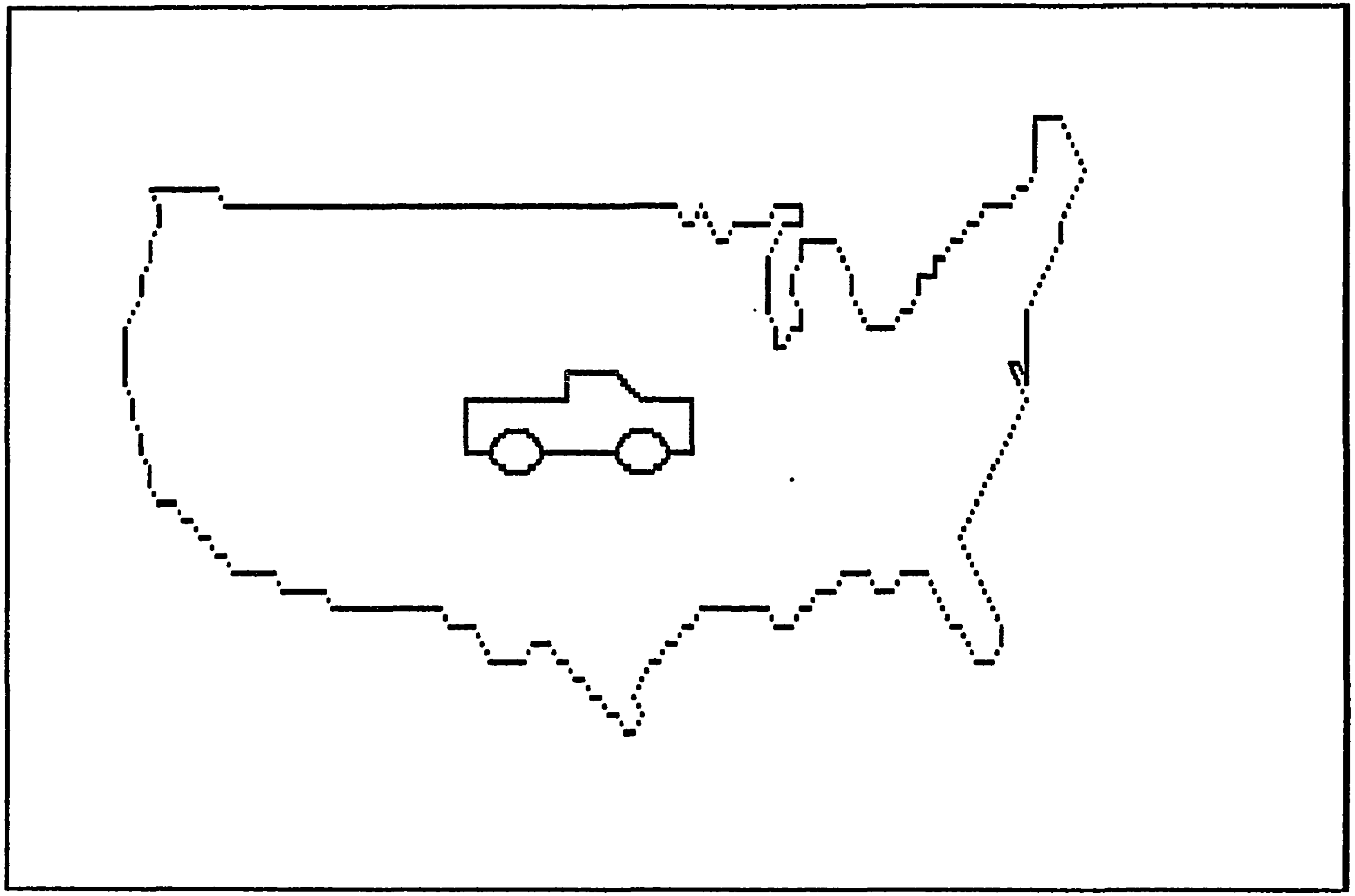


Figure 12.38. Image usal

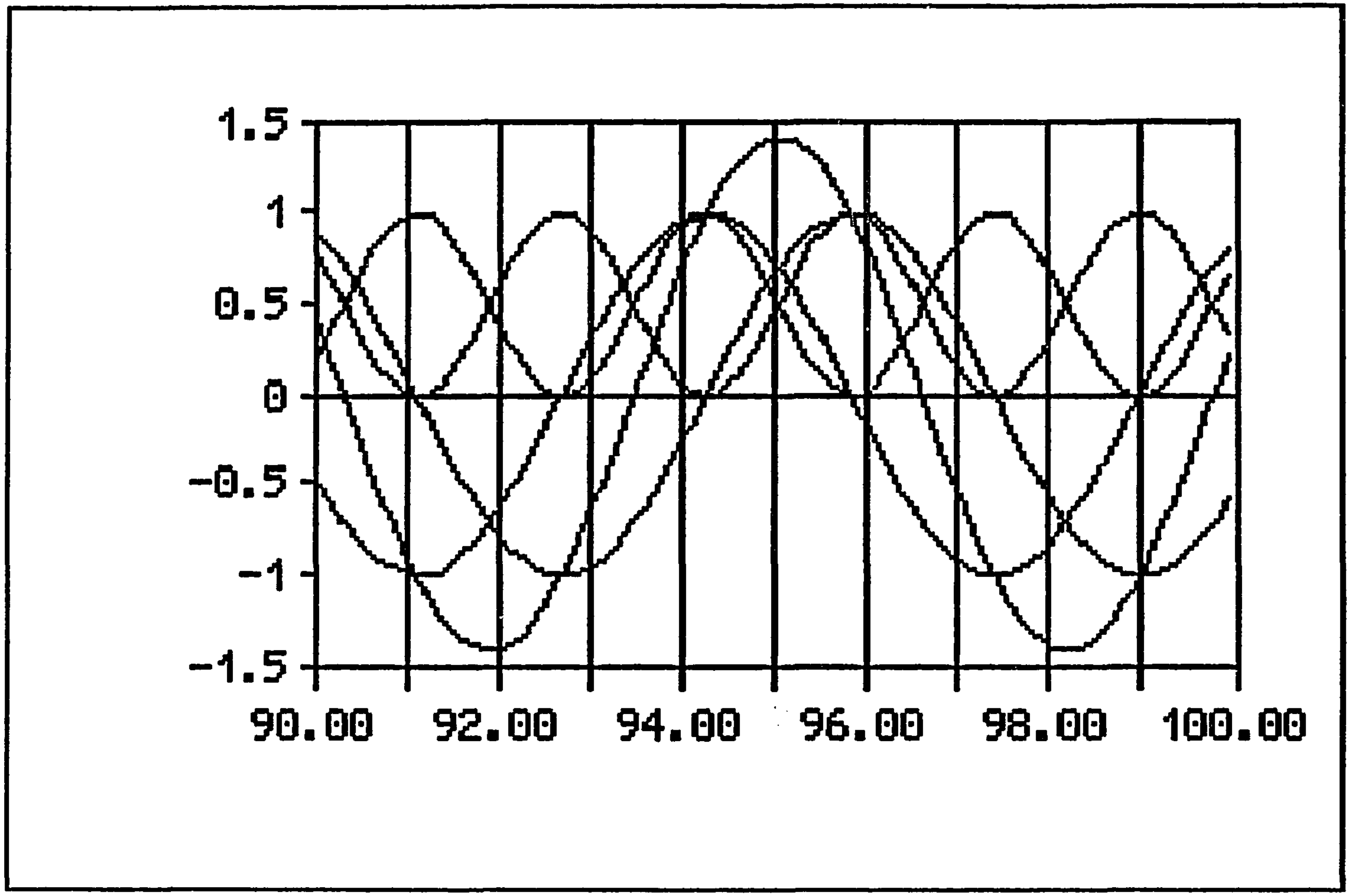


Figure 12.39. Image lotssin

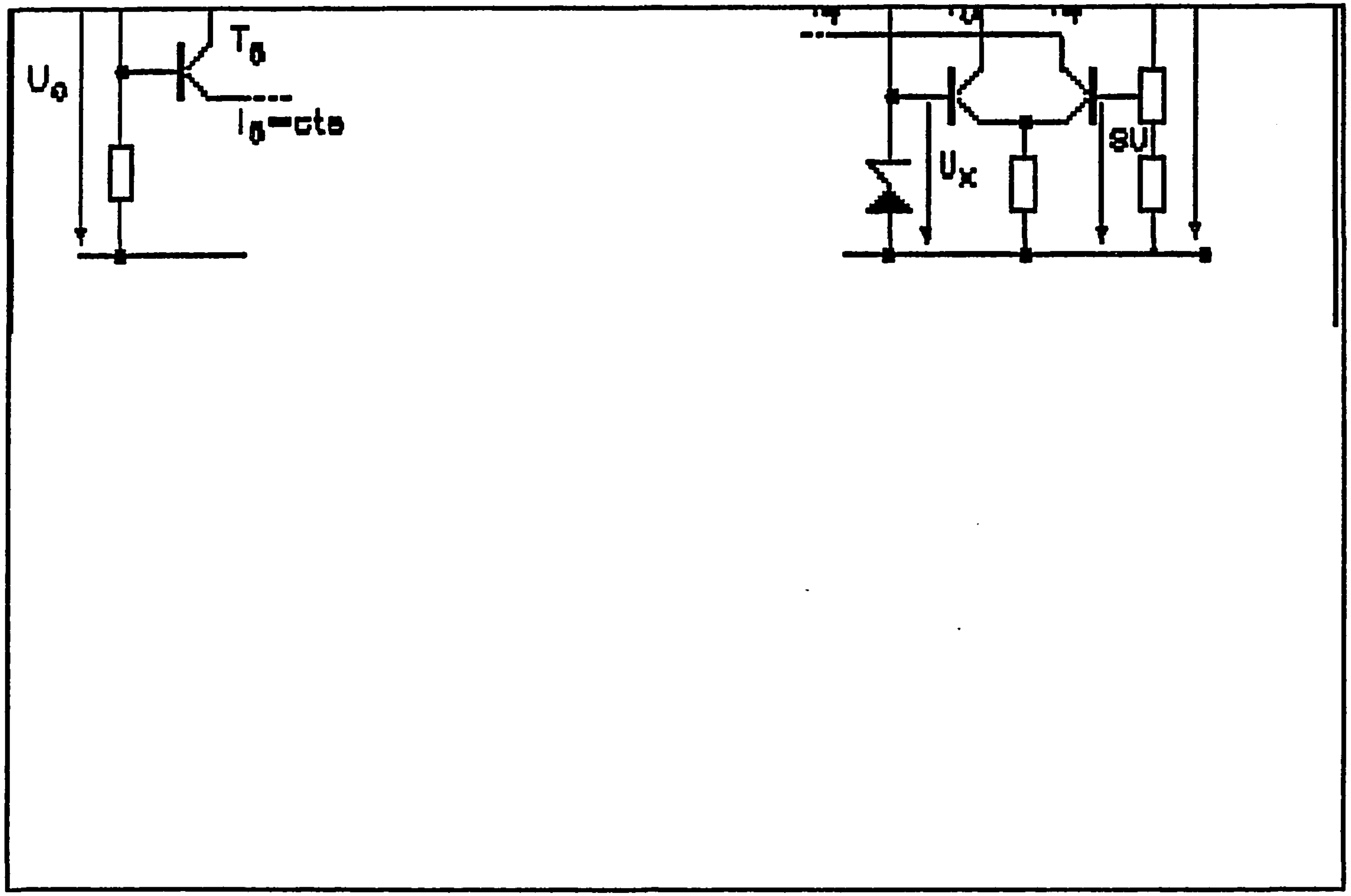


Figure 12.40. Image frnch3b

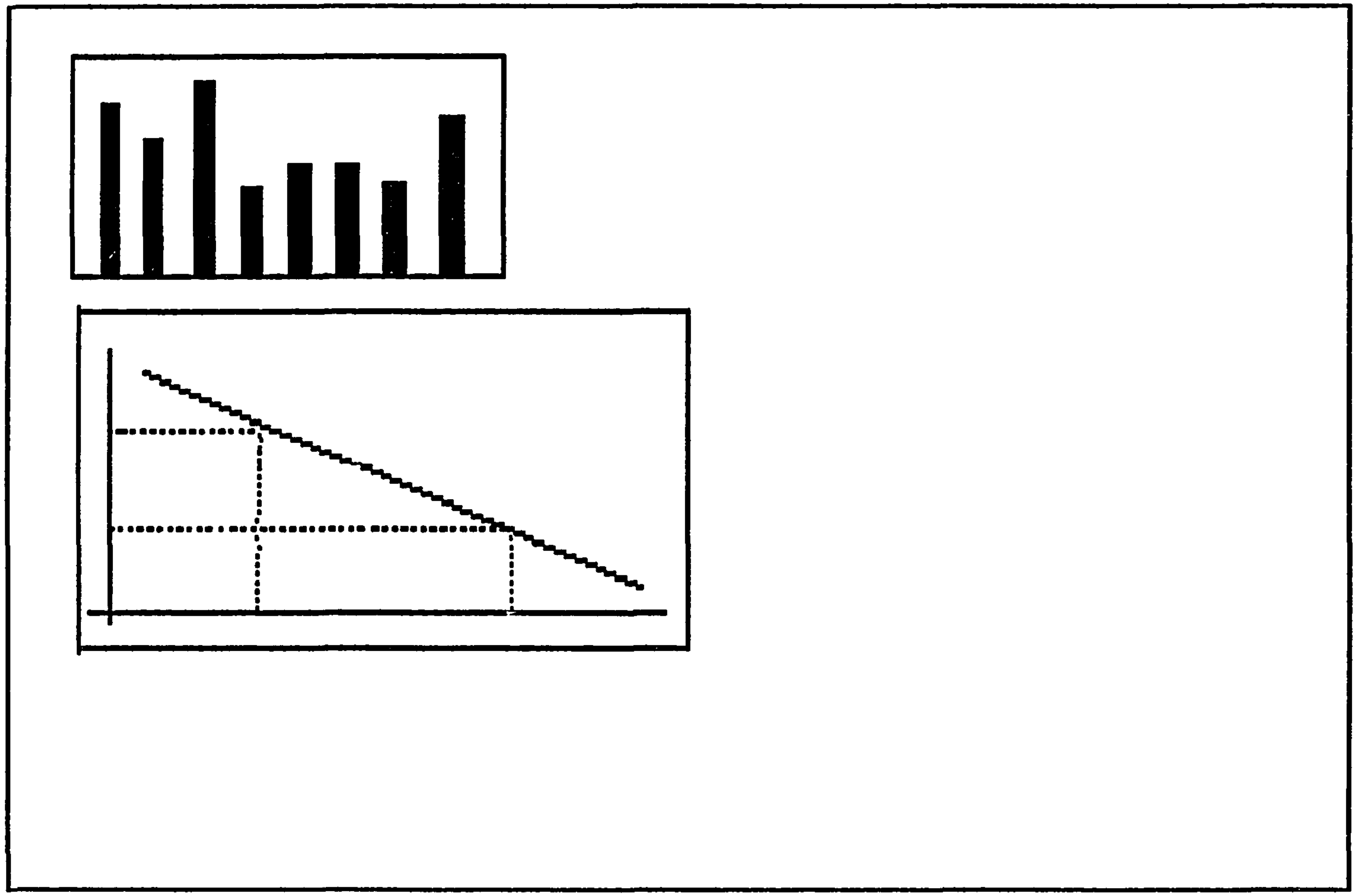


Figure 12.41. Image barchrt

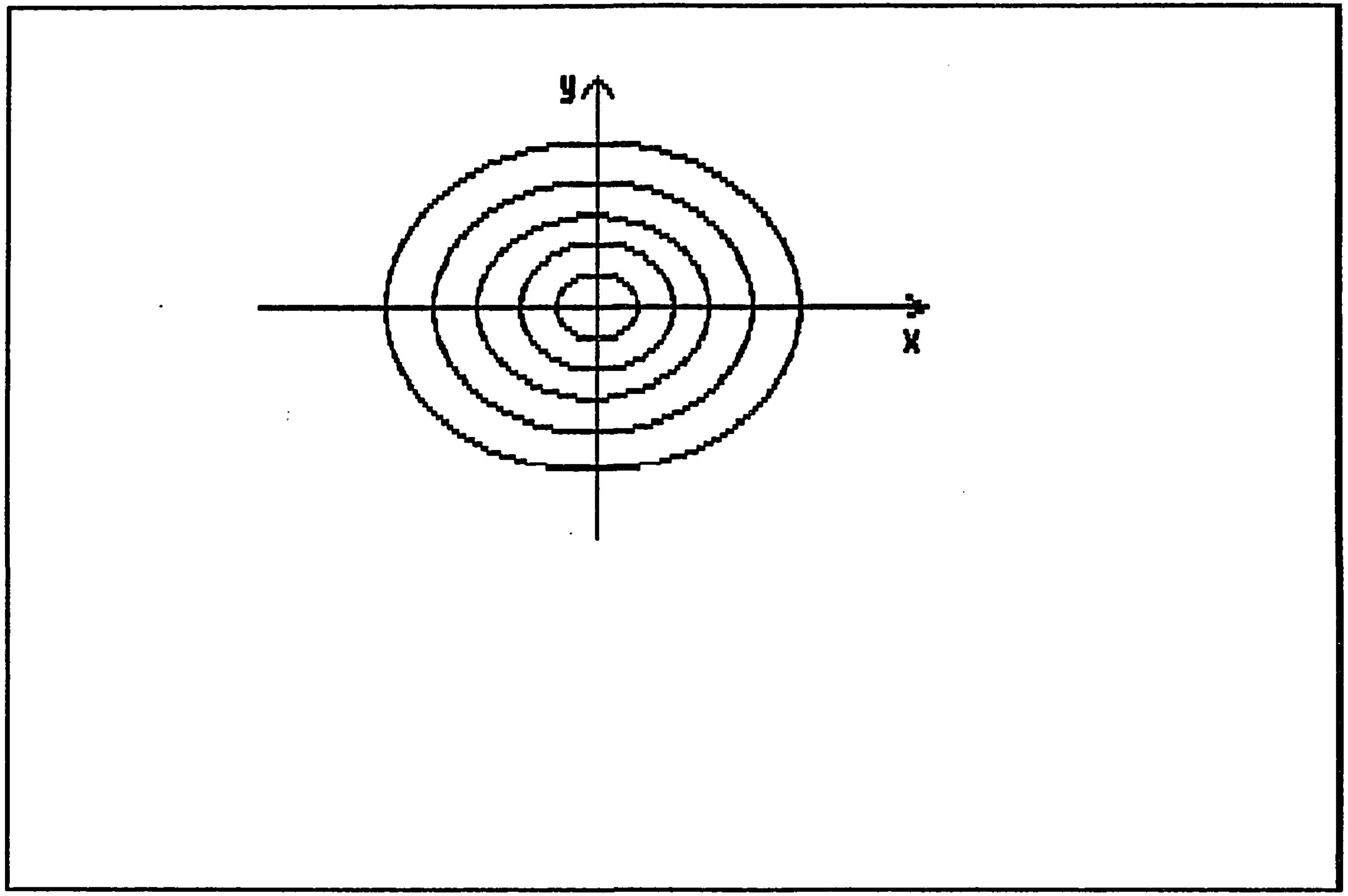


Figure 12.42. Image test2

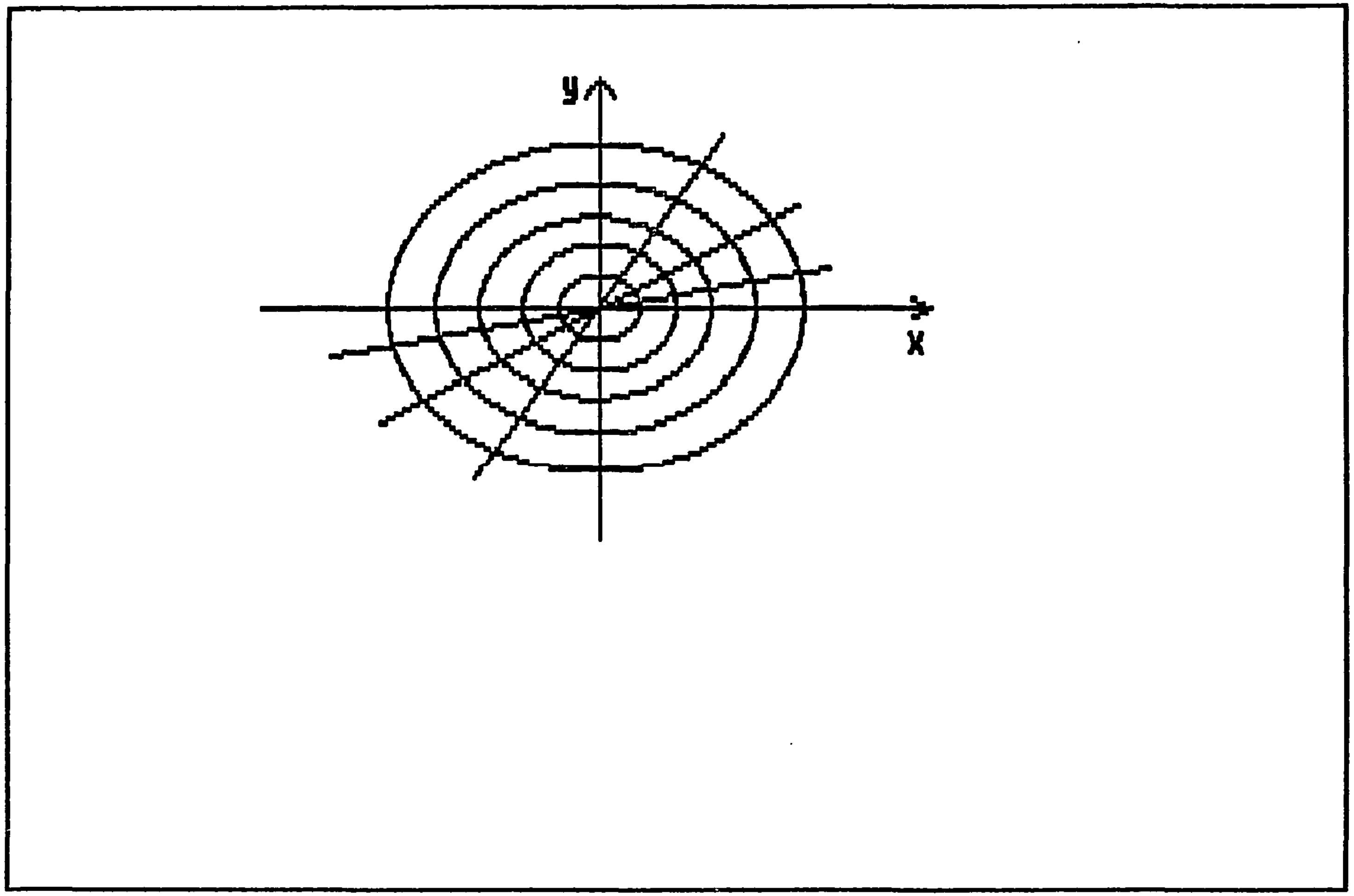


Figure 12.43. Image test3

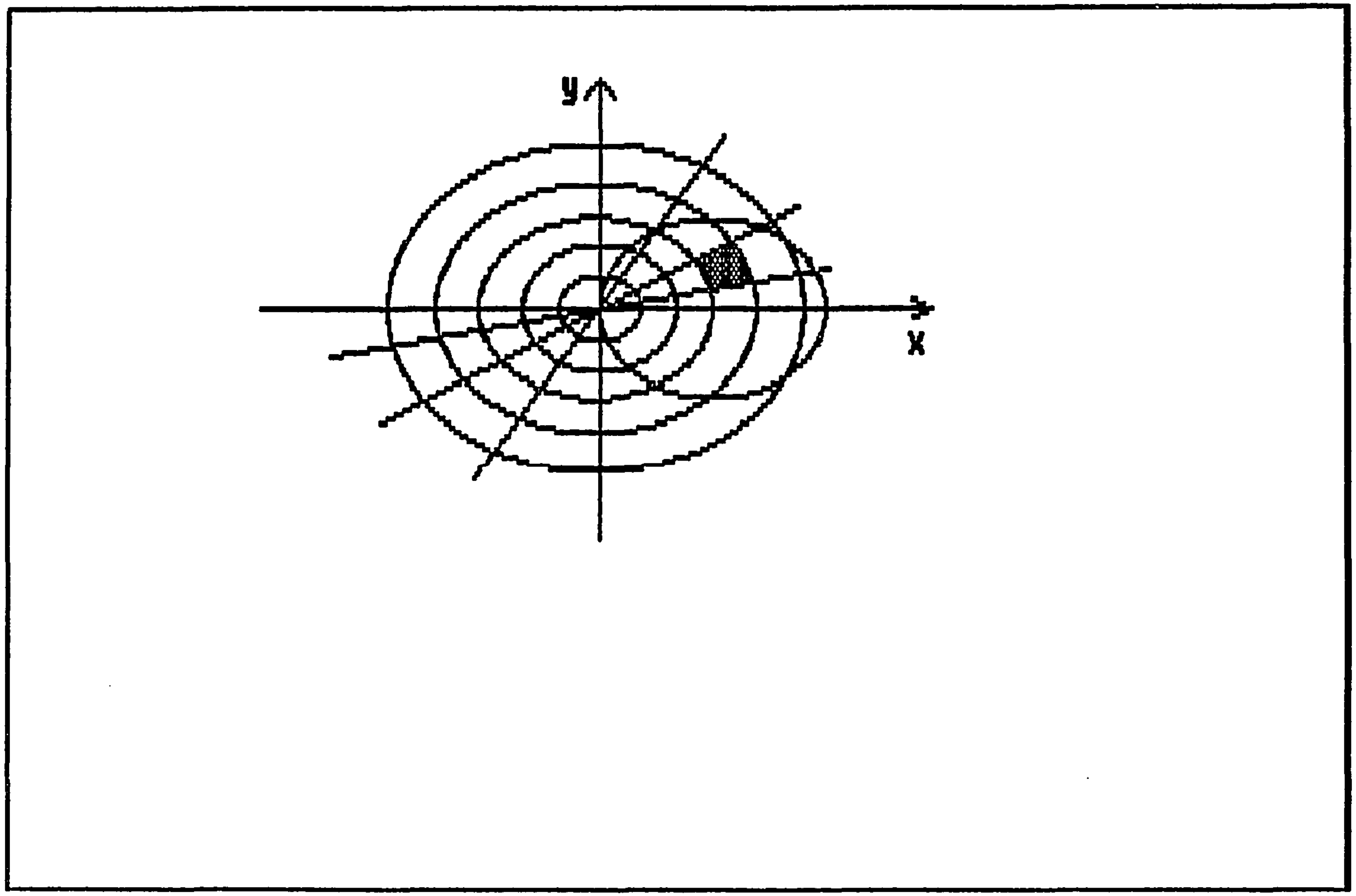


Figure 12.44. Image test4

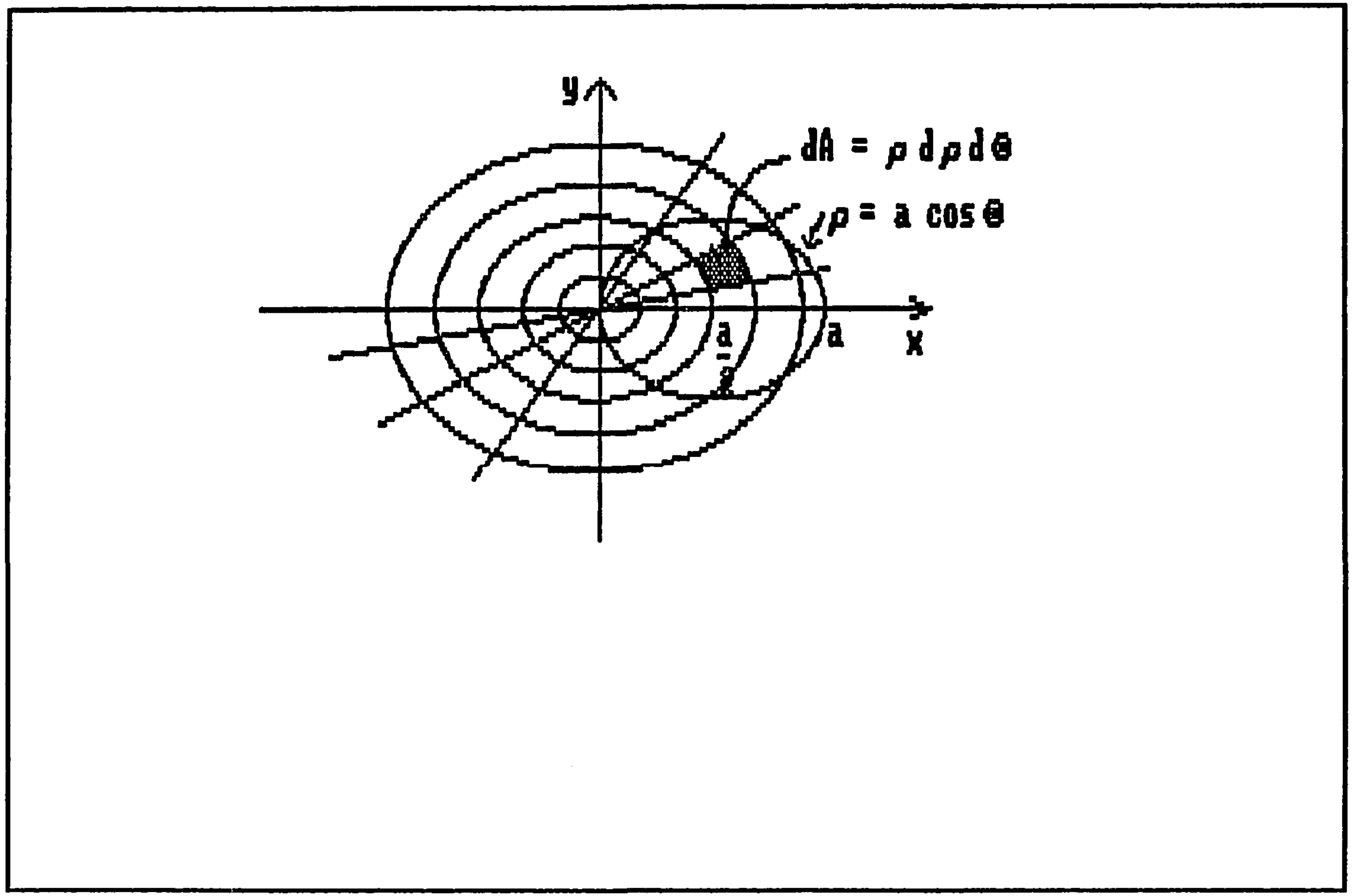


Figure 12.45. Image test5

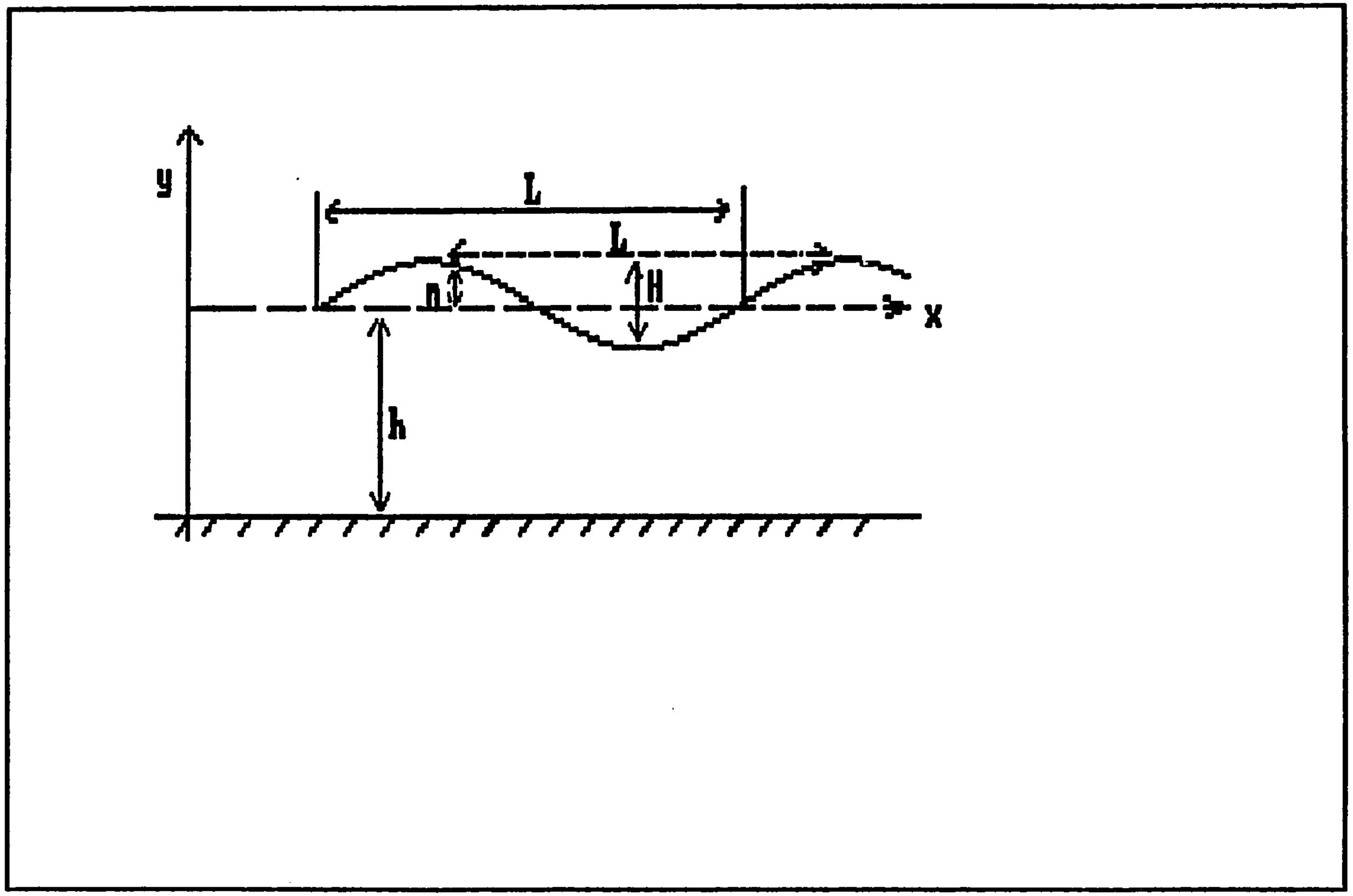


Figure 12.46. Image diagl

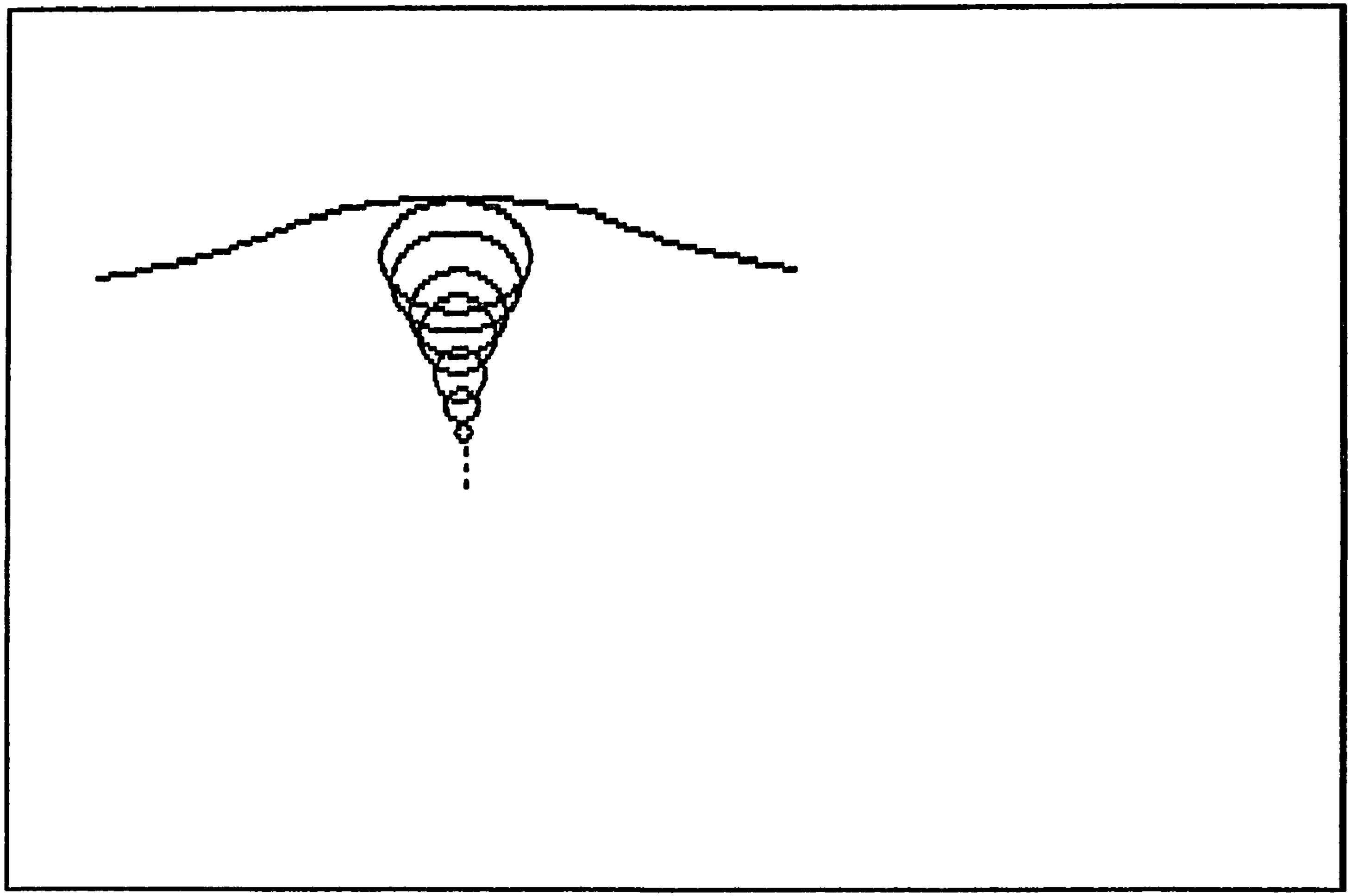


Figure 12.47. Image diag2

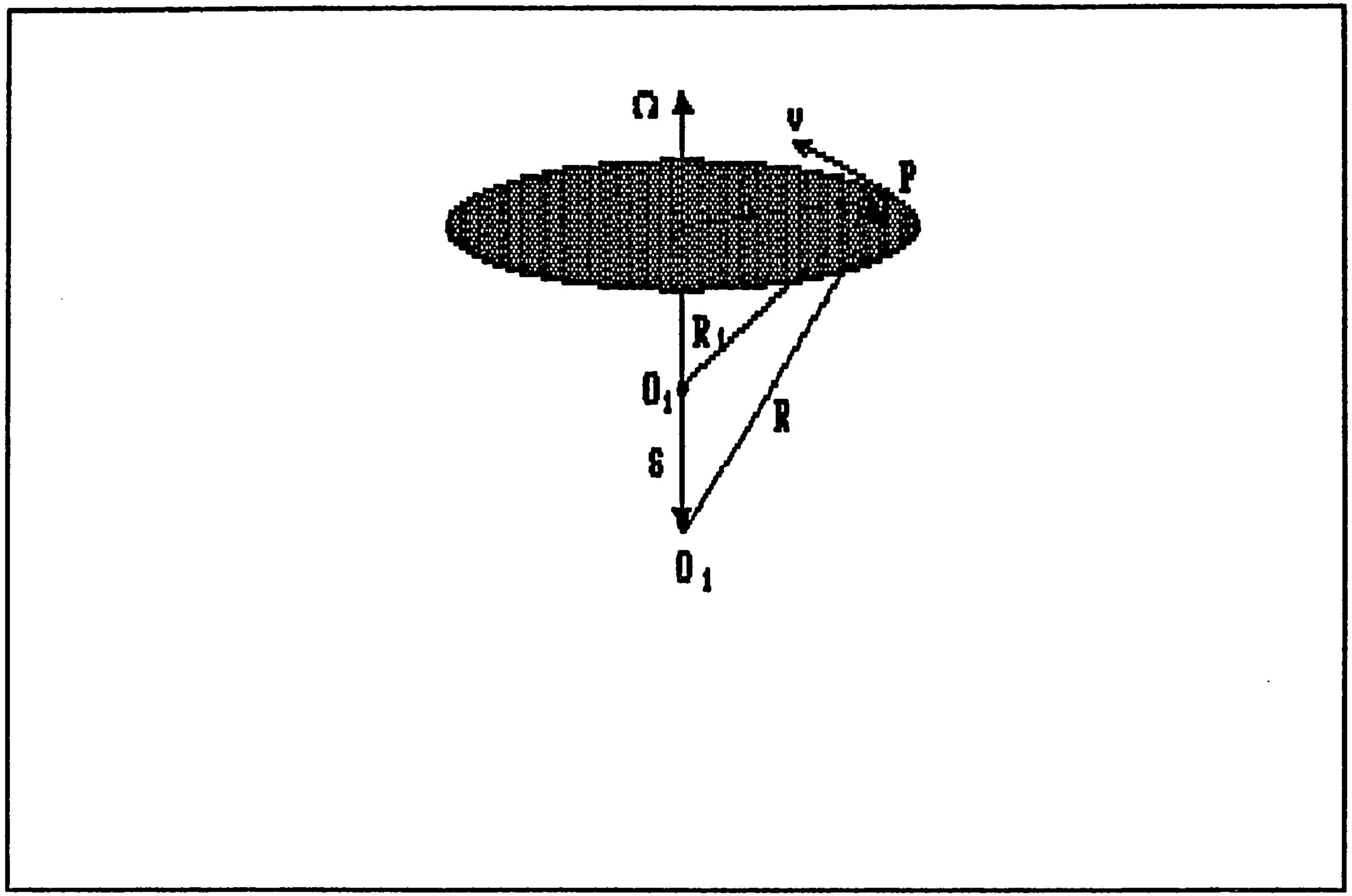


Figure 12.48. Image diag3

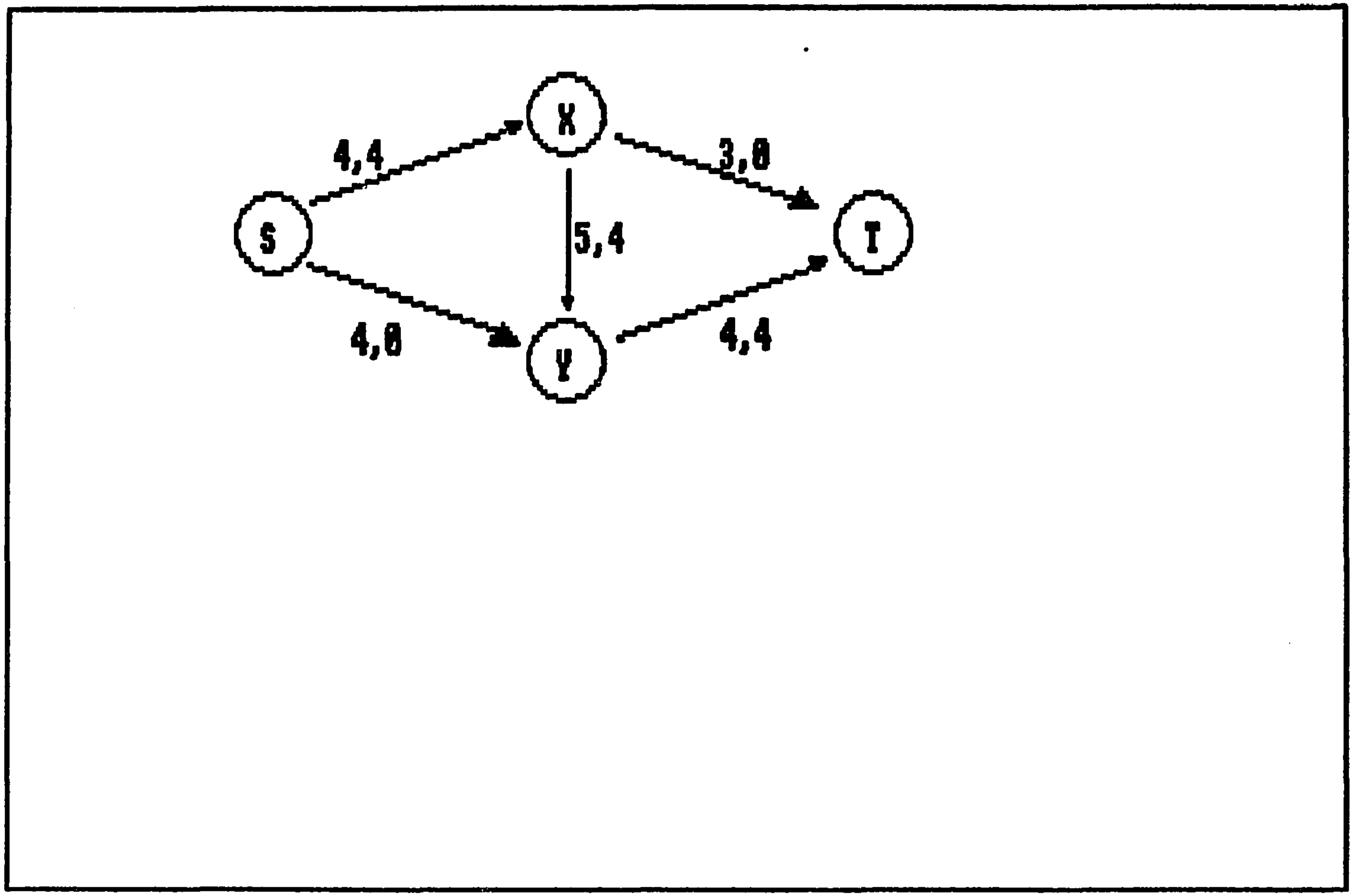


Figure 12.49. Image diag4

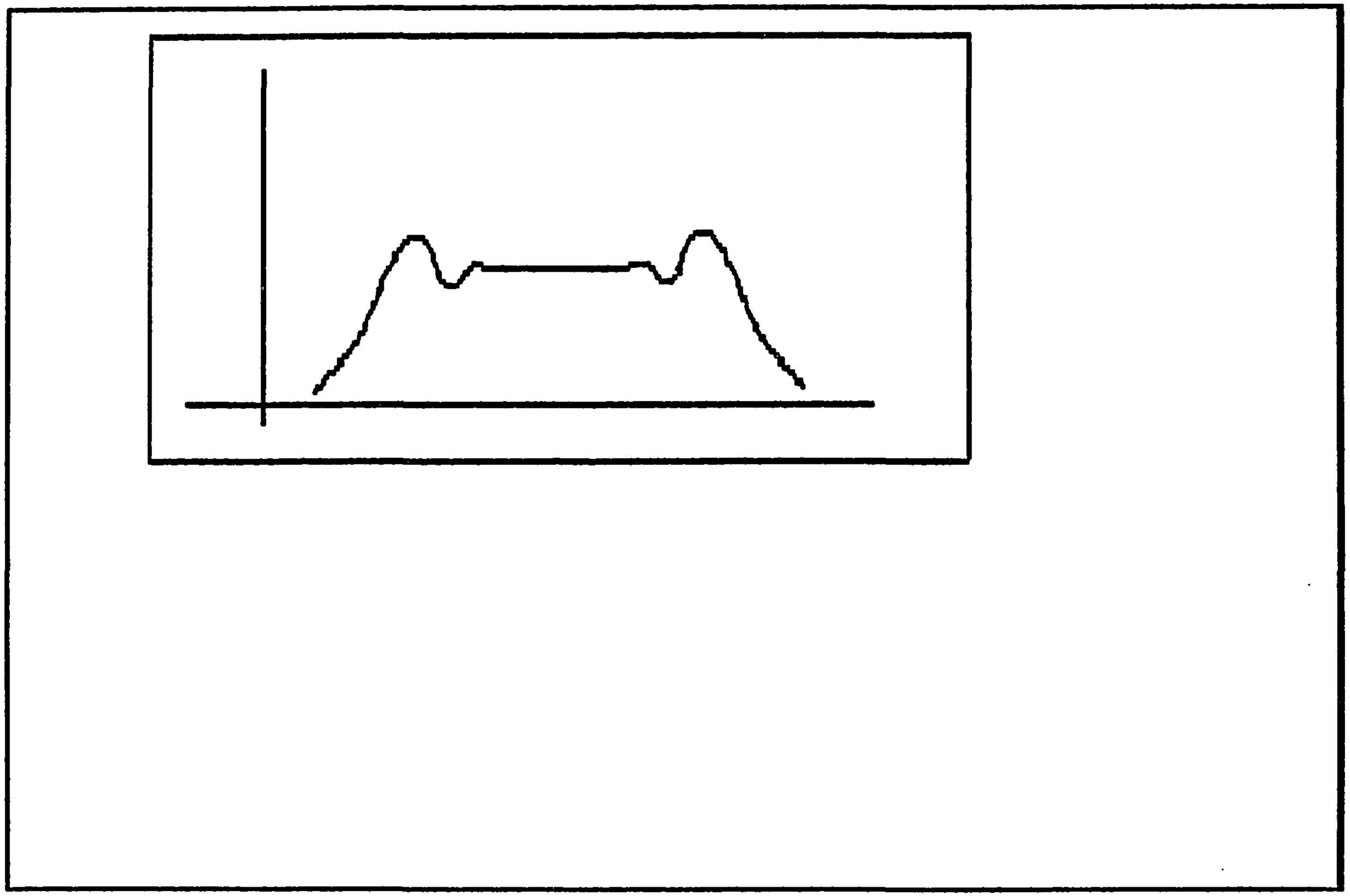


Figure 12.50. Image diag5

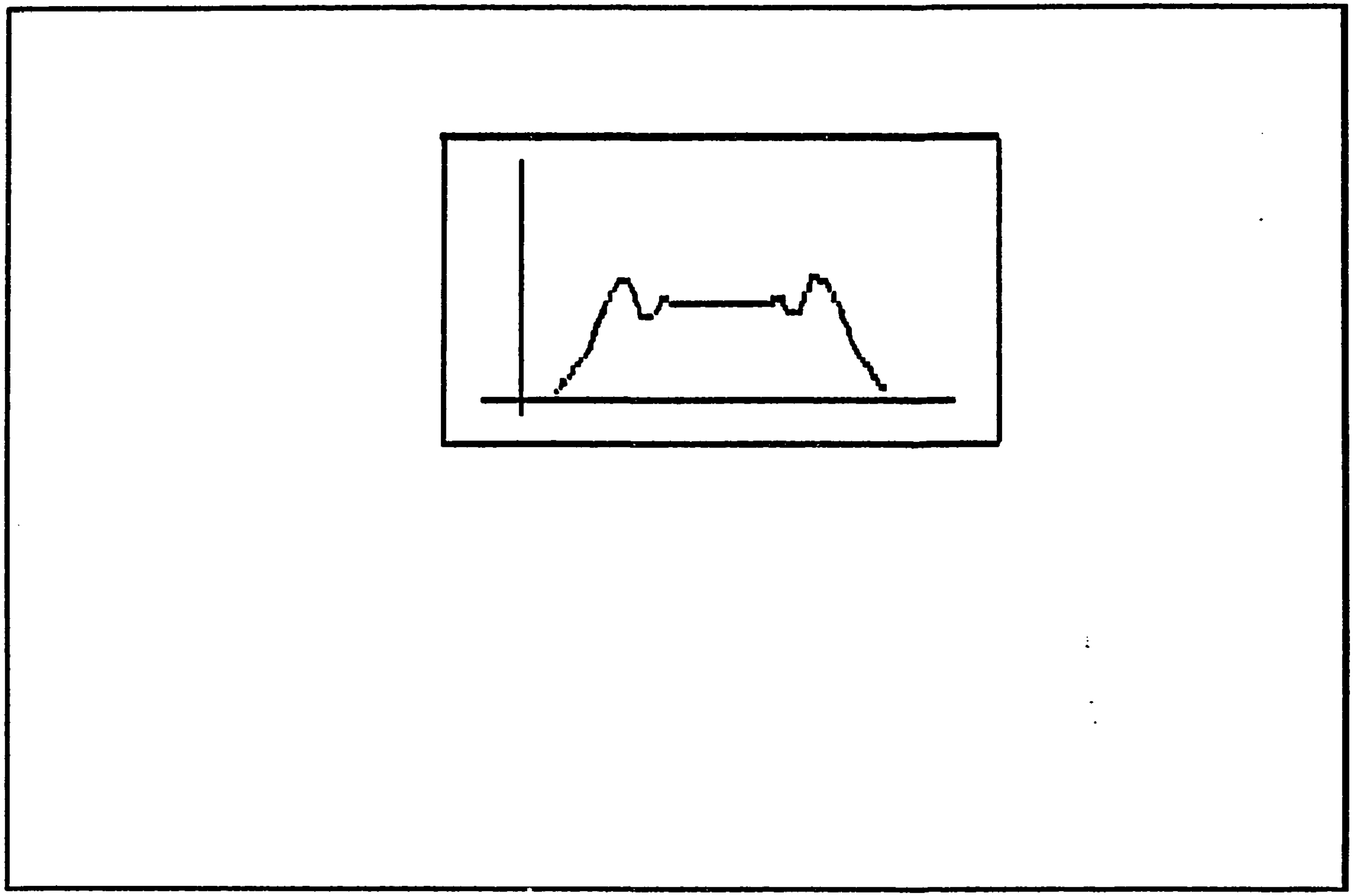


Figure 12.51. Image diag5s

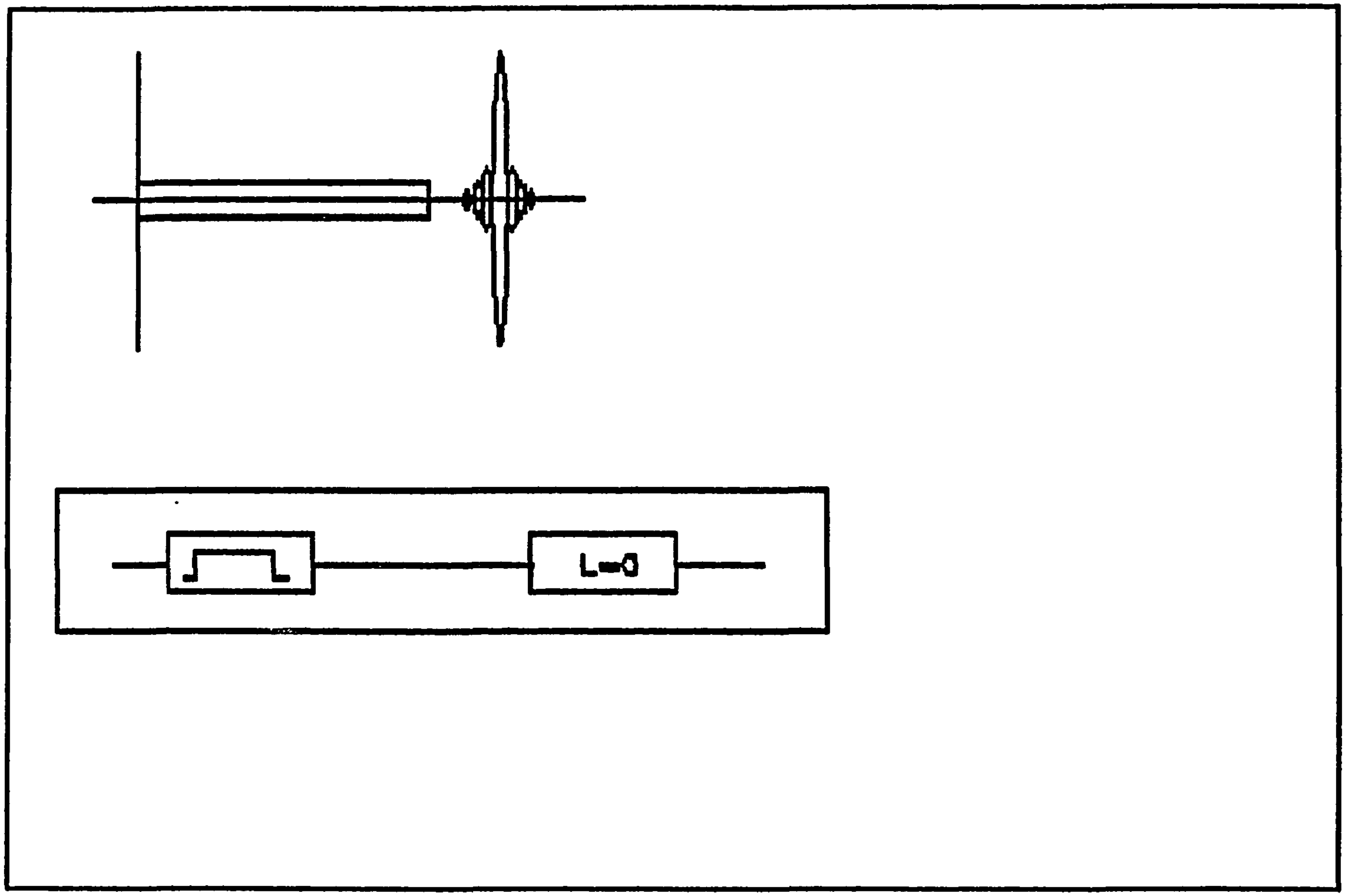


Figure 12.52. Image diag6

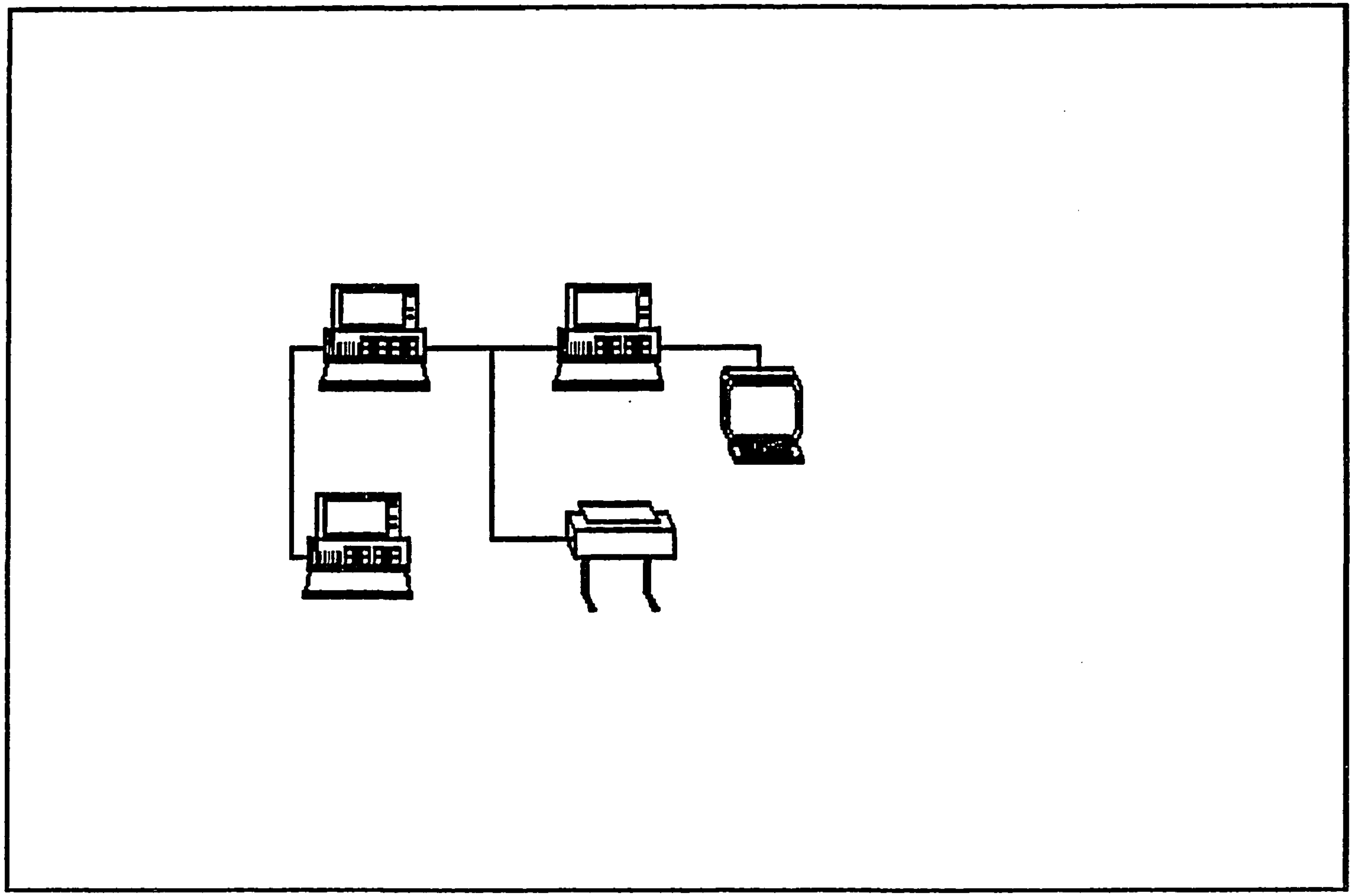
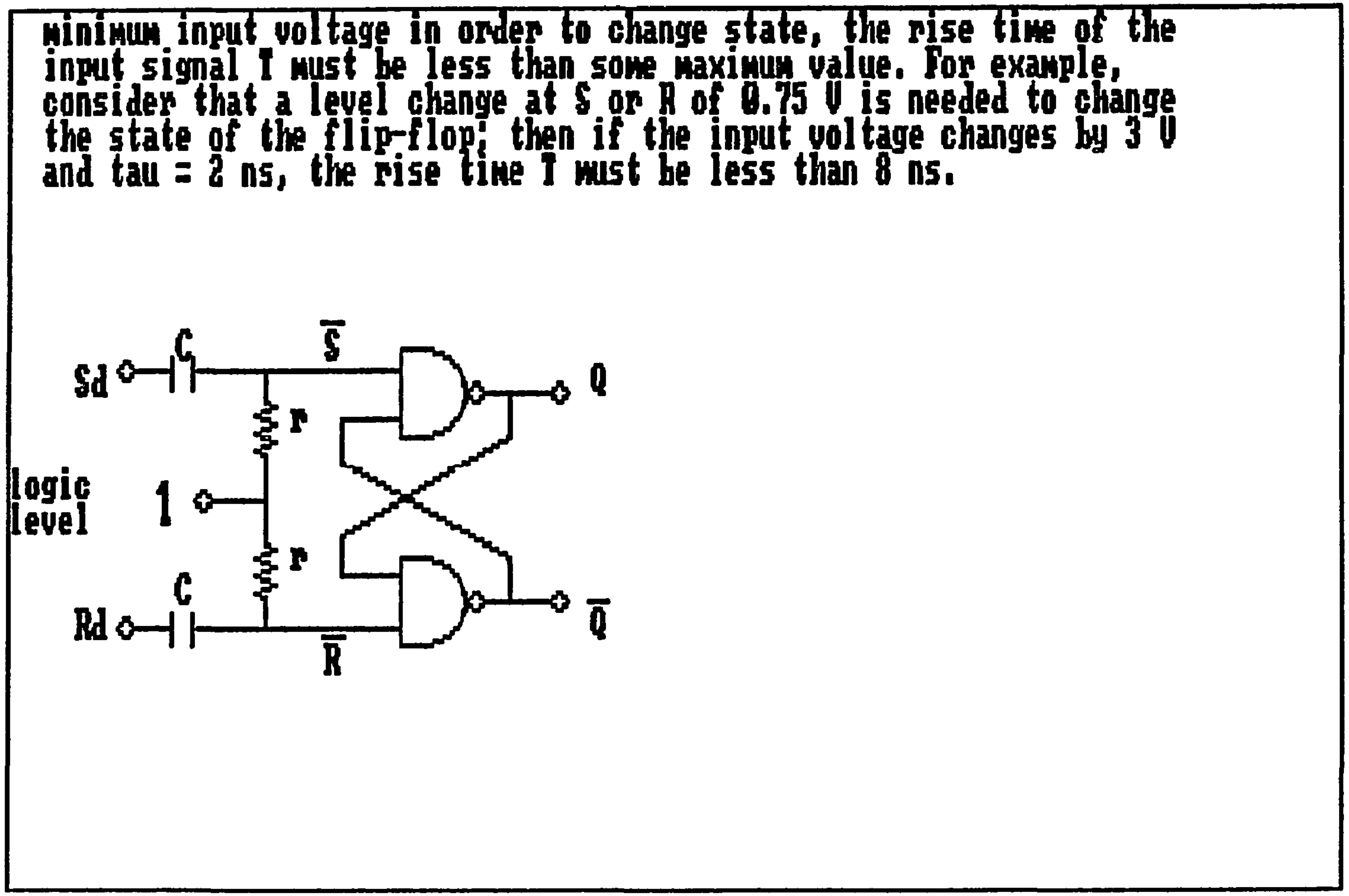


Figure 12.53. Image netwrk2



minimum input voltage in order to change state, the rise time of the input signal T must be less than some maximum value. For example, consider that a level change at S or R of 0.75 V is needed to change the state of the flip-flop; then if the input voltage changes by 3 V and tau = 2 ns, the rise time T must be less than 8 ns.

Figure 12.54. Image pdrawl

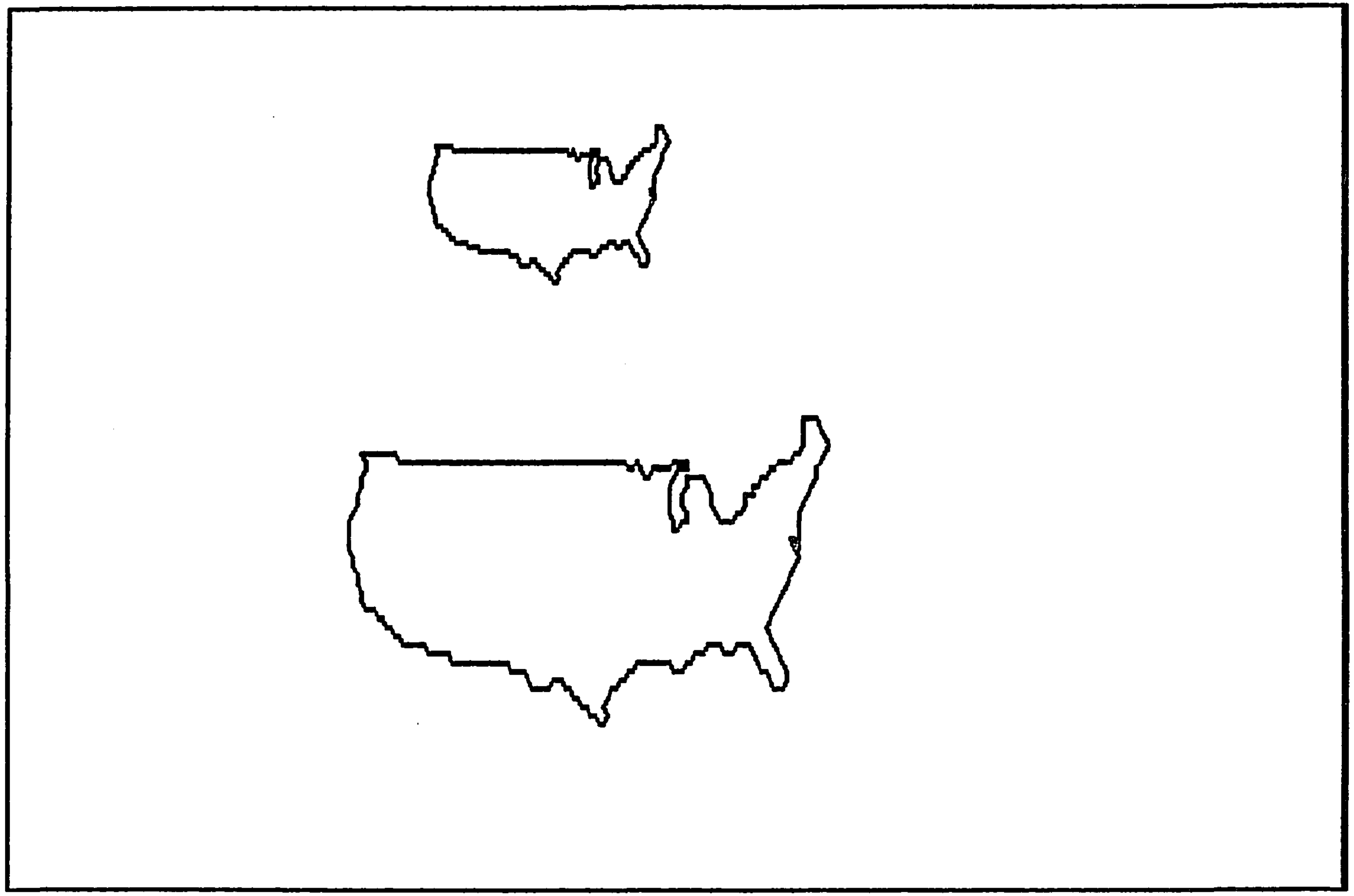
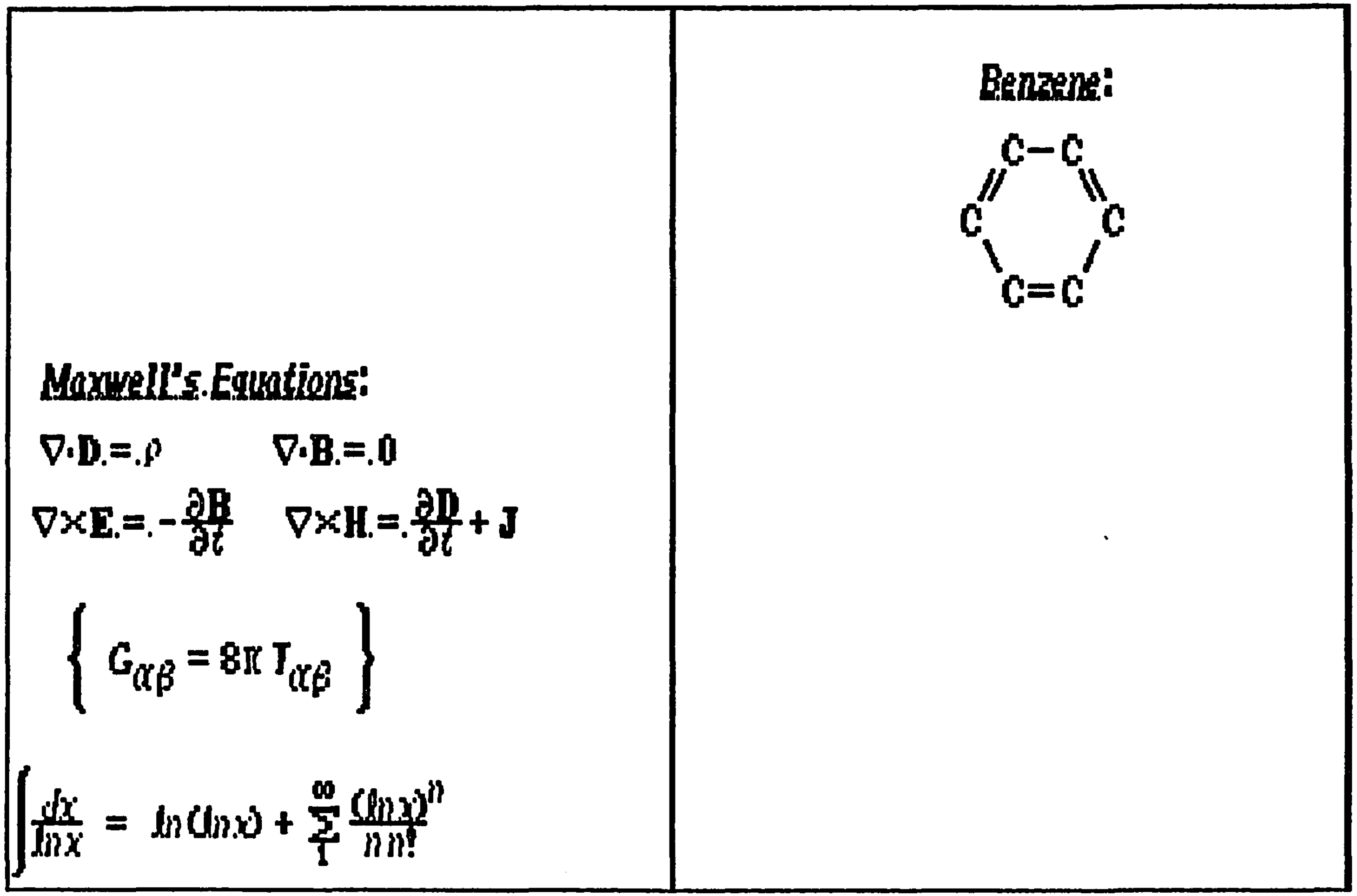


Figure 12.55. Image usa2



Maxwell's Equations:

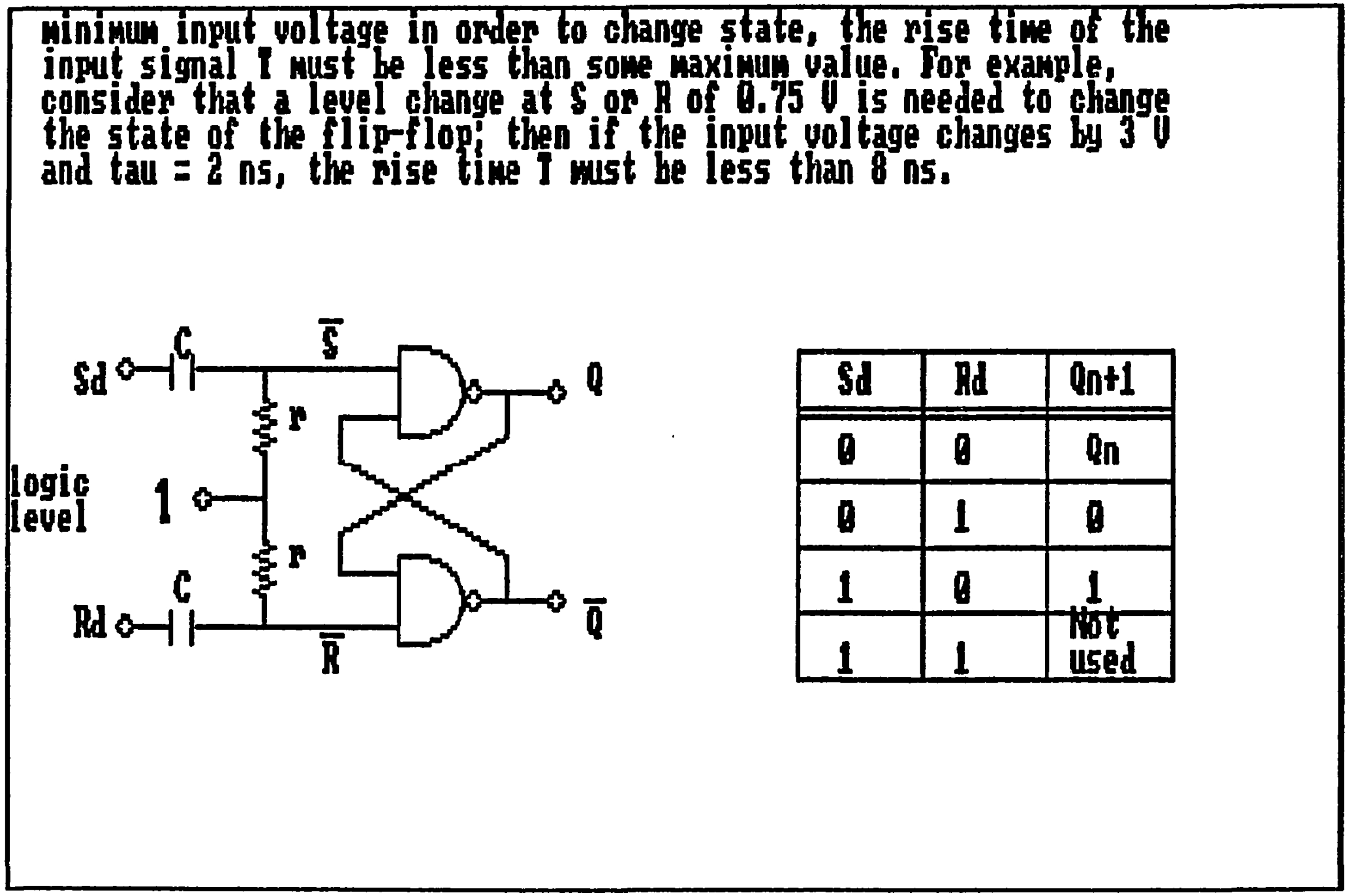
$$\nabla\cdot\mathbf{D} = \rho \qquad \nabla\cdot\mathbf{B} = 0$$

$$\nabla\times\mathbf{E} = -\frac{\partial\mathbf{B}}{\partial t} \qquad \nabla\times\mathbf{H} = \frac{\partial\mathbf{D}}{\partial t} + \mathbf{J}$$

$$\left\{ G_{\alpha\beta} = 8\pi\, T_{\alpha\beta} \right\}$$

$$\int \frac{dx}{\ln x} = \ln(\ln x) + \sum_{1}^{\infty} \frac{(\ln x)^n}{n\,n!}$$

Figure 12.56. Image science3



minimum input voltage in order to change state, the rise time of the input signal T must be less than some maximum value. For example, consider that a level change at S or R of 0.75 V is needed to change the state of the flip-flop; then if the input voltage changes by 3 V and tau = 2 ns, the rise time T must be less than 8 ns.

Sd	Rd	Qn+1
0	0	Qn
0	1	0
1	0	1
1	1	Not used

Figure 12.57. Image pdraw2

Figure 12.58. Image bignames

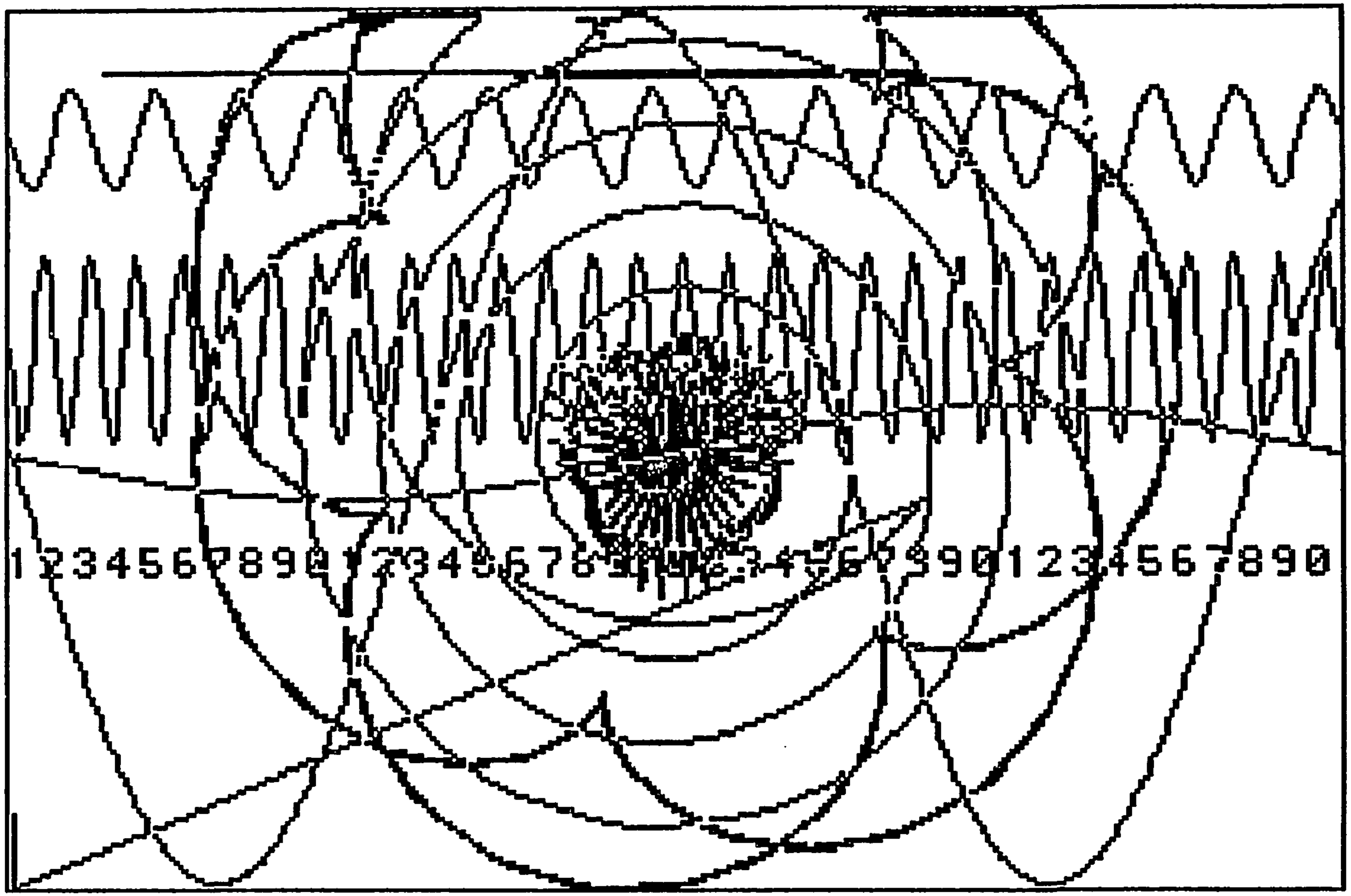


Figure 12.59. Image sun

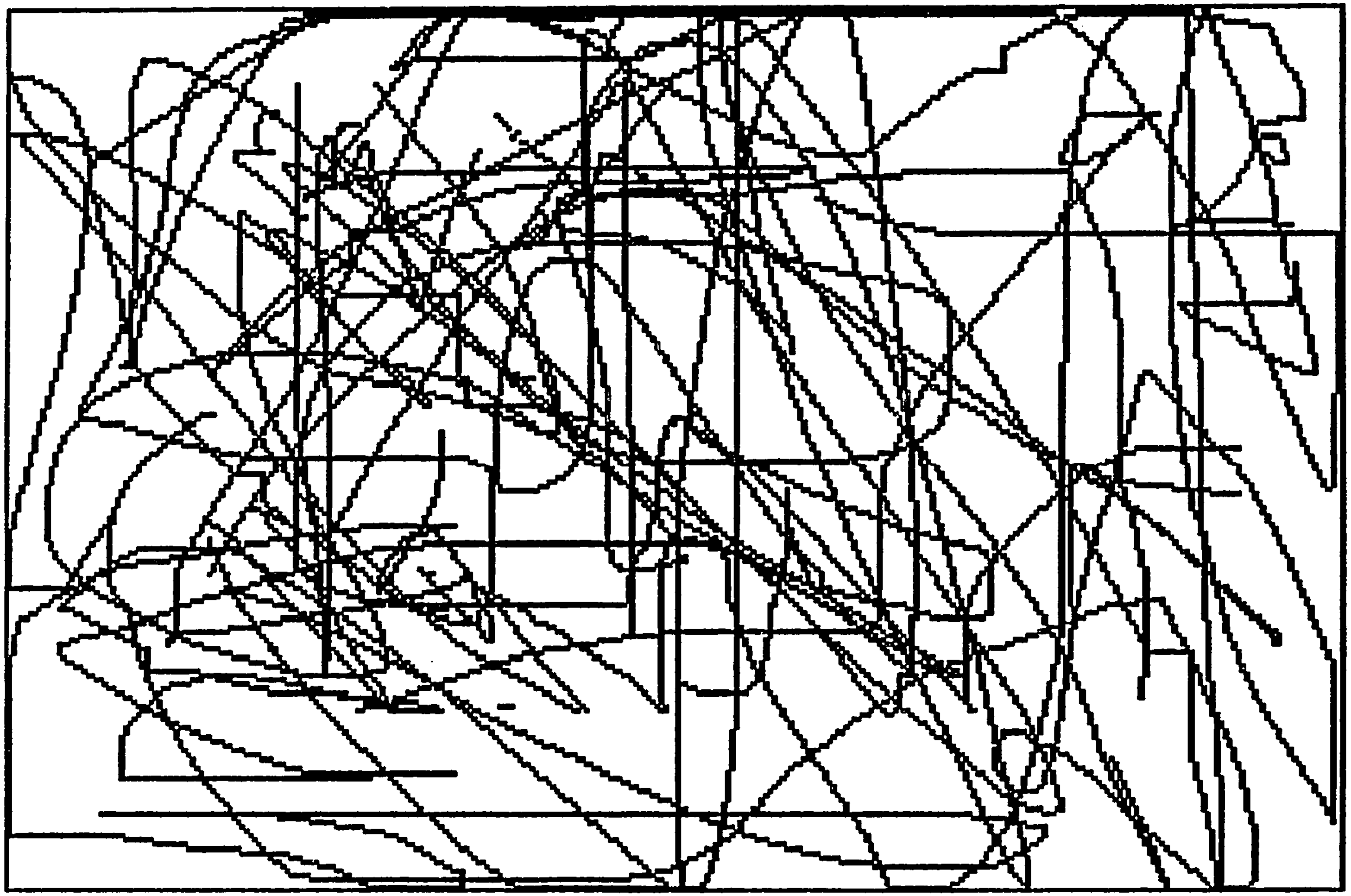
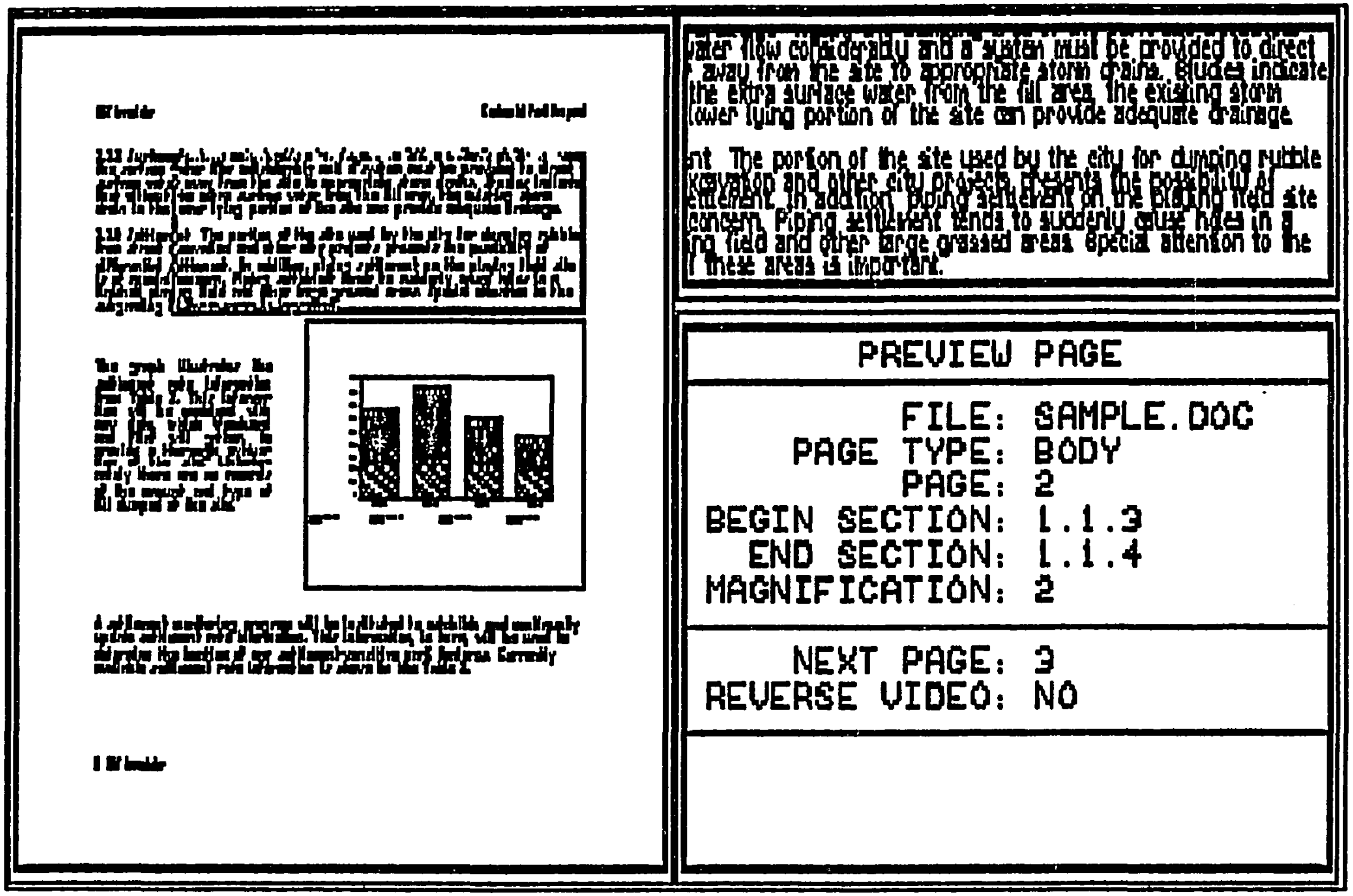


Figure 12.60. Image hazard



ater flow considerably and a system must be provided to direct away from the site to appropriate storm drains. Studies indicate the extra surface water from the fill area, the existing storm lower lying portion of the site can provide adequate drainage.

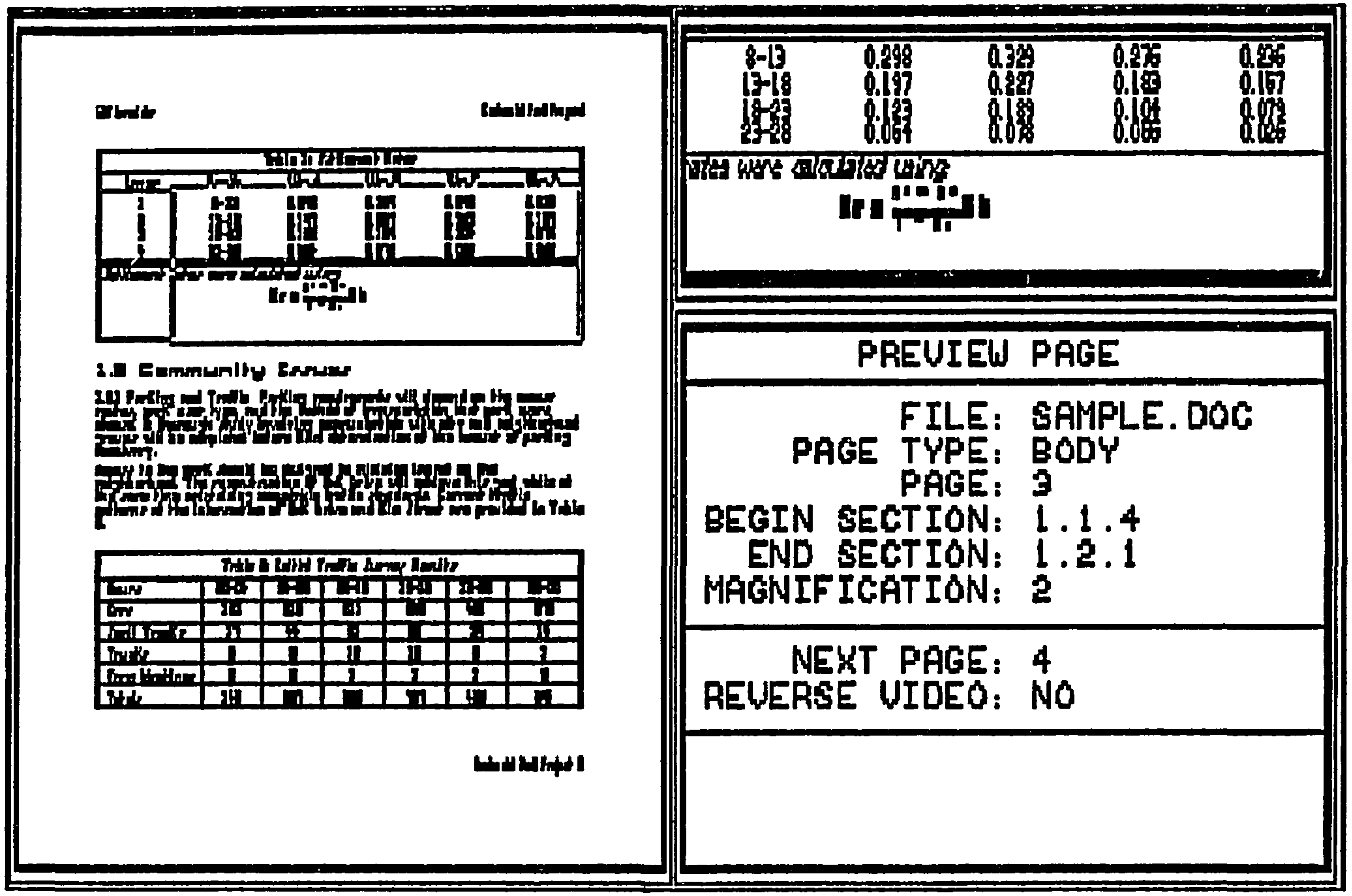
nt The portion of the site used by the city for dumping rubble xcavation and other city projects presents the possibility of ettlement. In addition, piping settlement on the playing field site concern. Piping settlement tends to suddenly cause holes in a ng field and other large grassed areas. Special attention to the these areas is important.

PREVIEW PAGE

FILE: SAMPLE.DOC
PAGE TYPE: BODY
PAGE: 2
BEGIN SECTION: 1.1.3
END SECTION: 1.1.4
MAGNIFICATION: 2

NEXT PAGE: 3
REVERSE VIDEO: NO

Figure 12.61. Image manscl



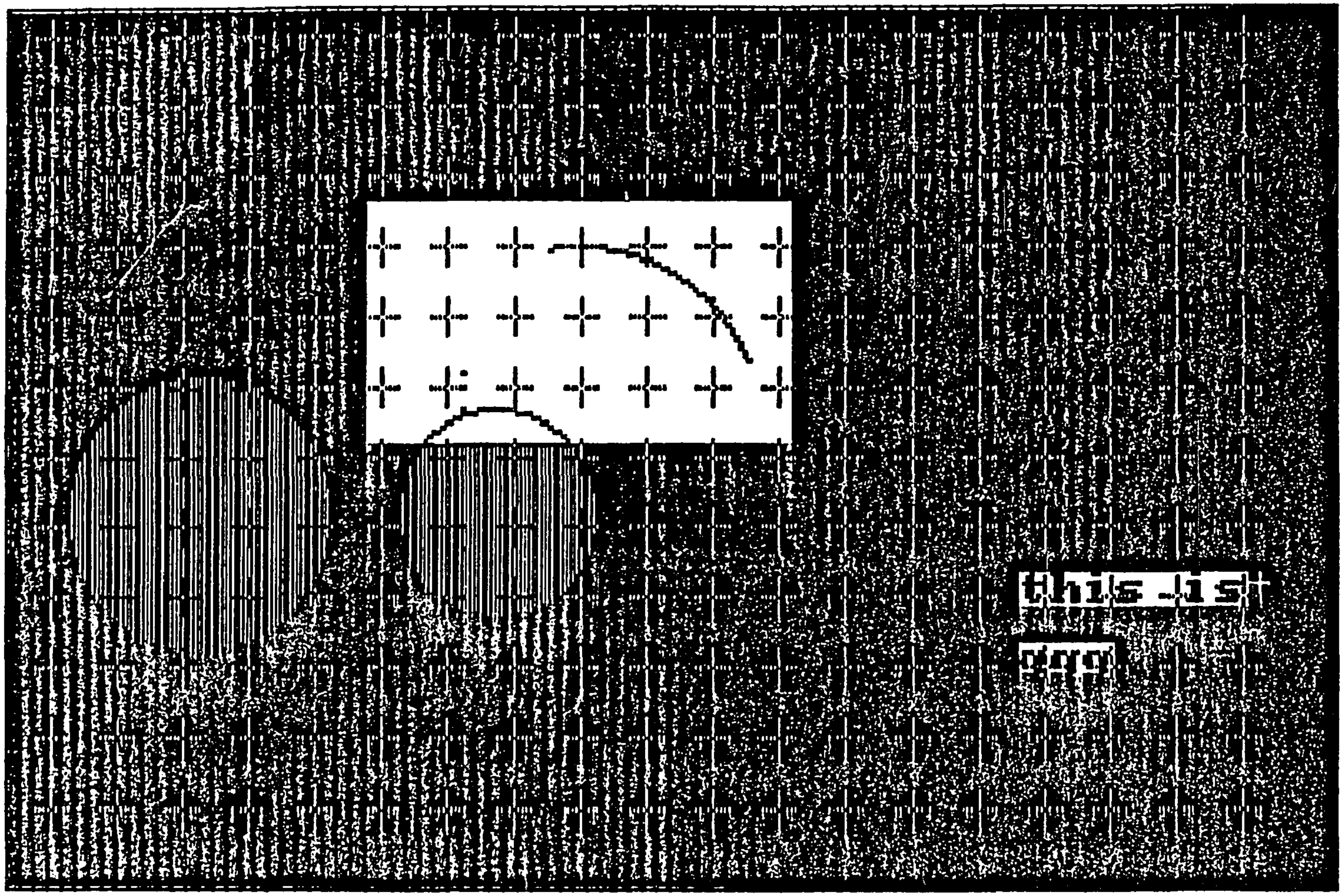


Figure 12.63. Image fig2

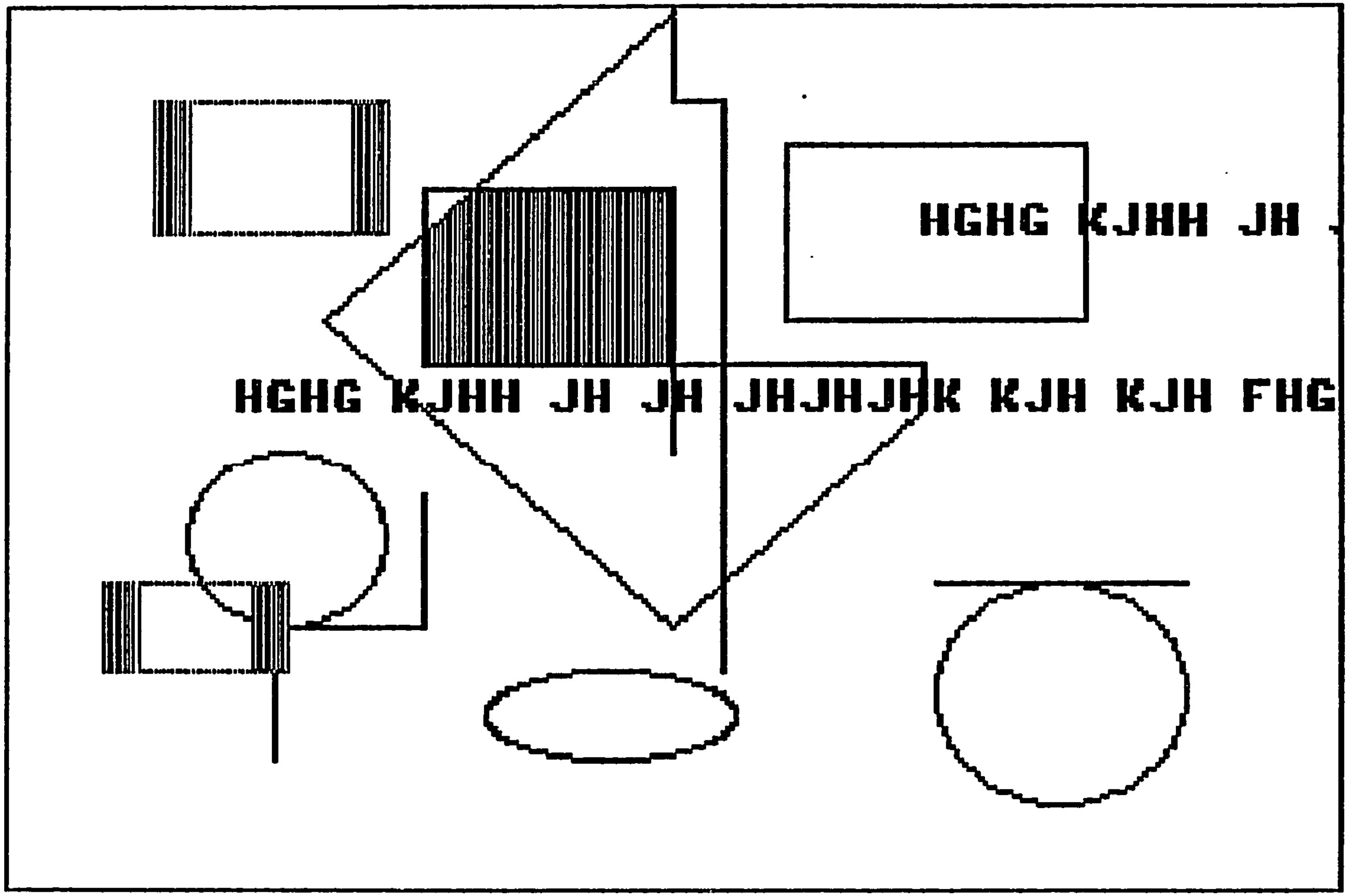


Figure 12.64. Image fig4

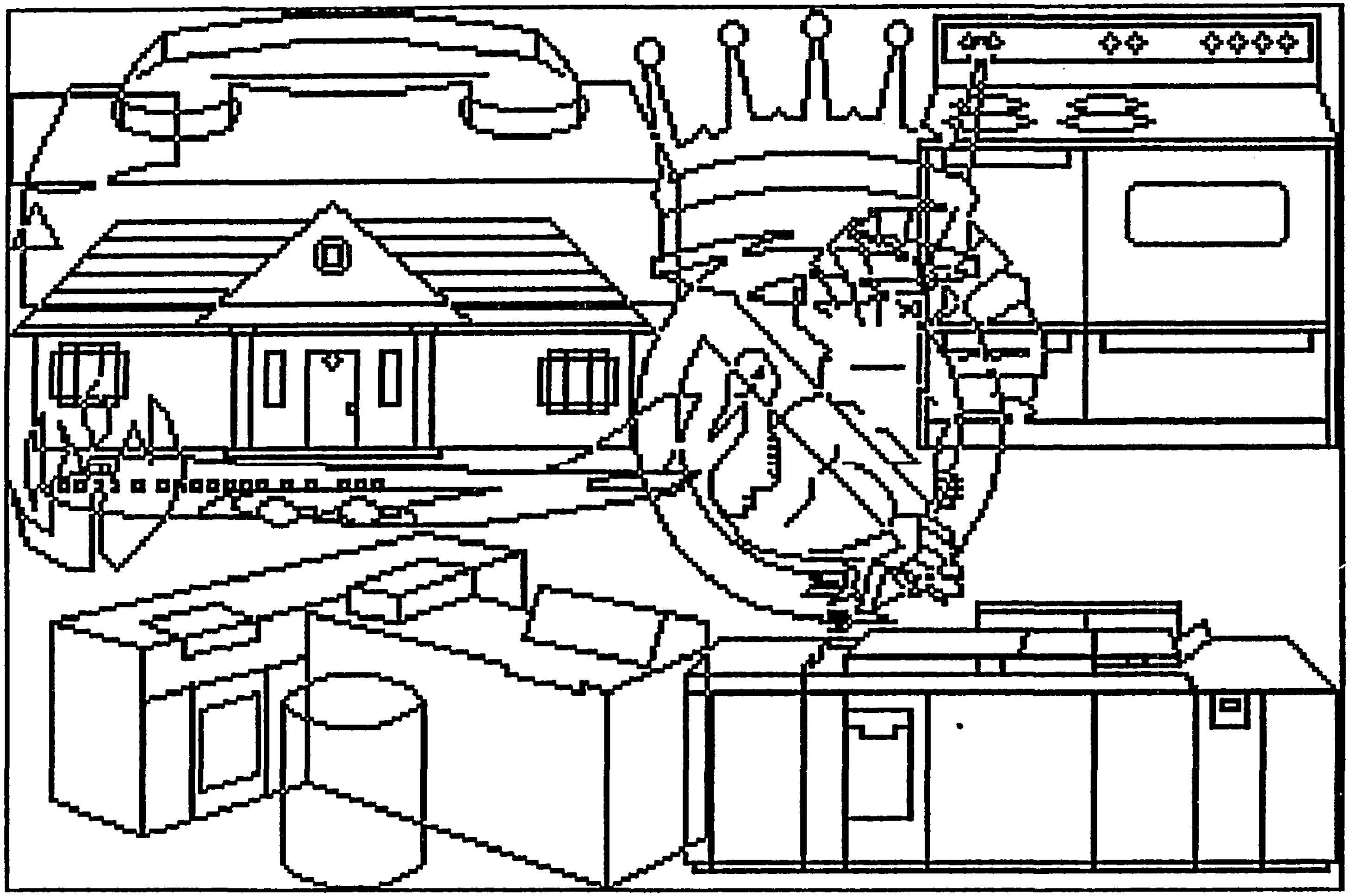


Figure 12.65. Image fig6

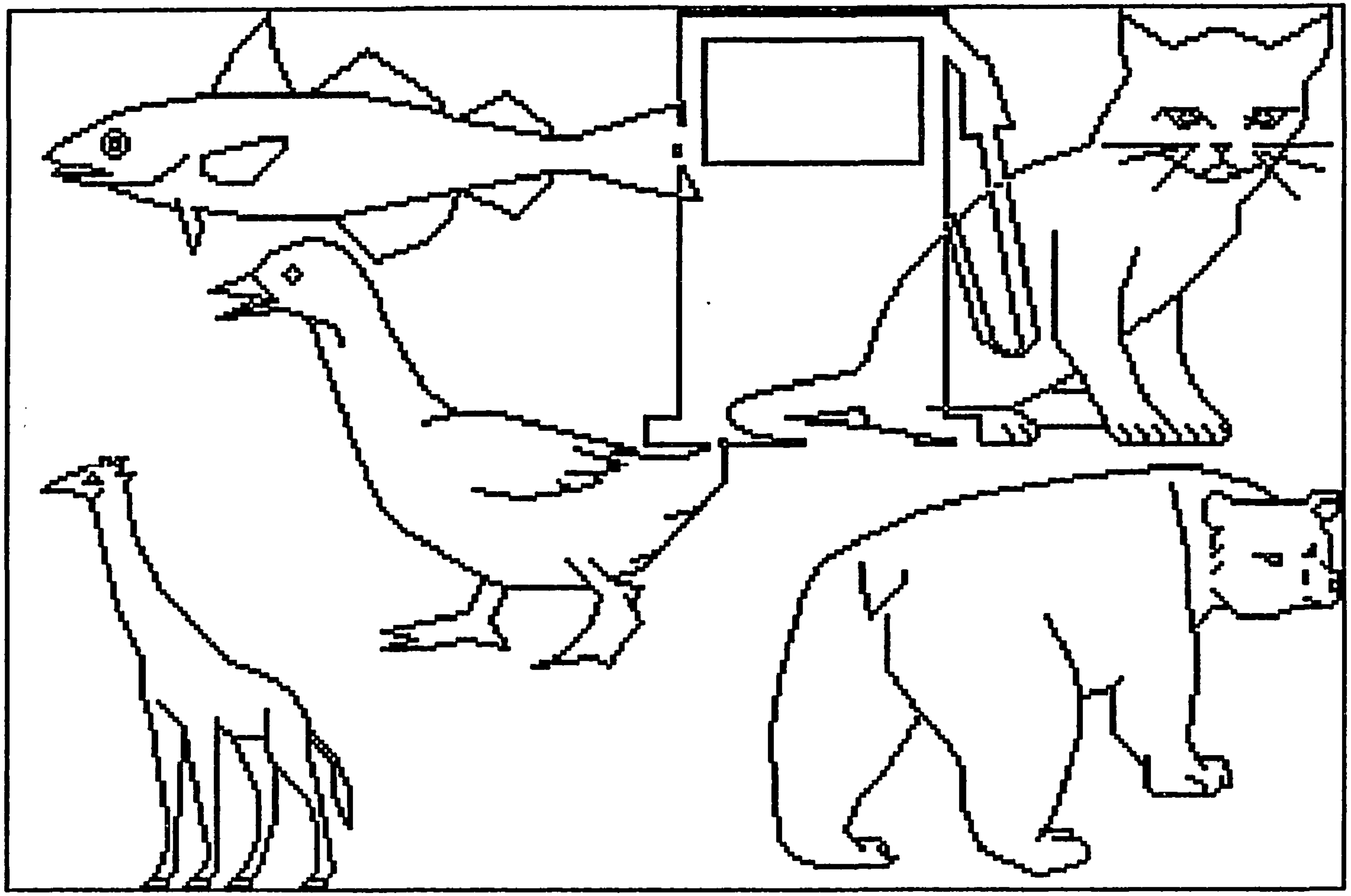


Figure 12.66. Image fig7

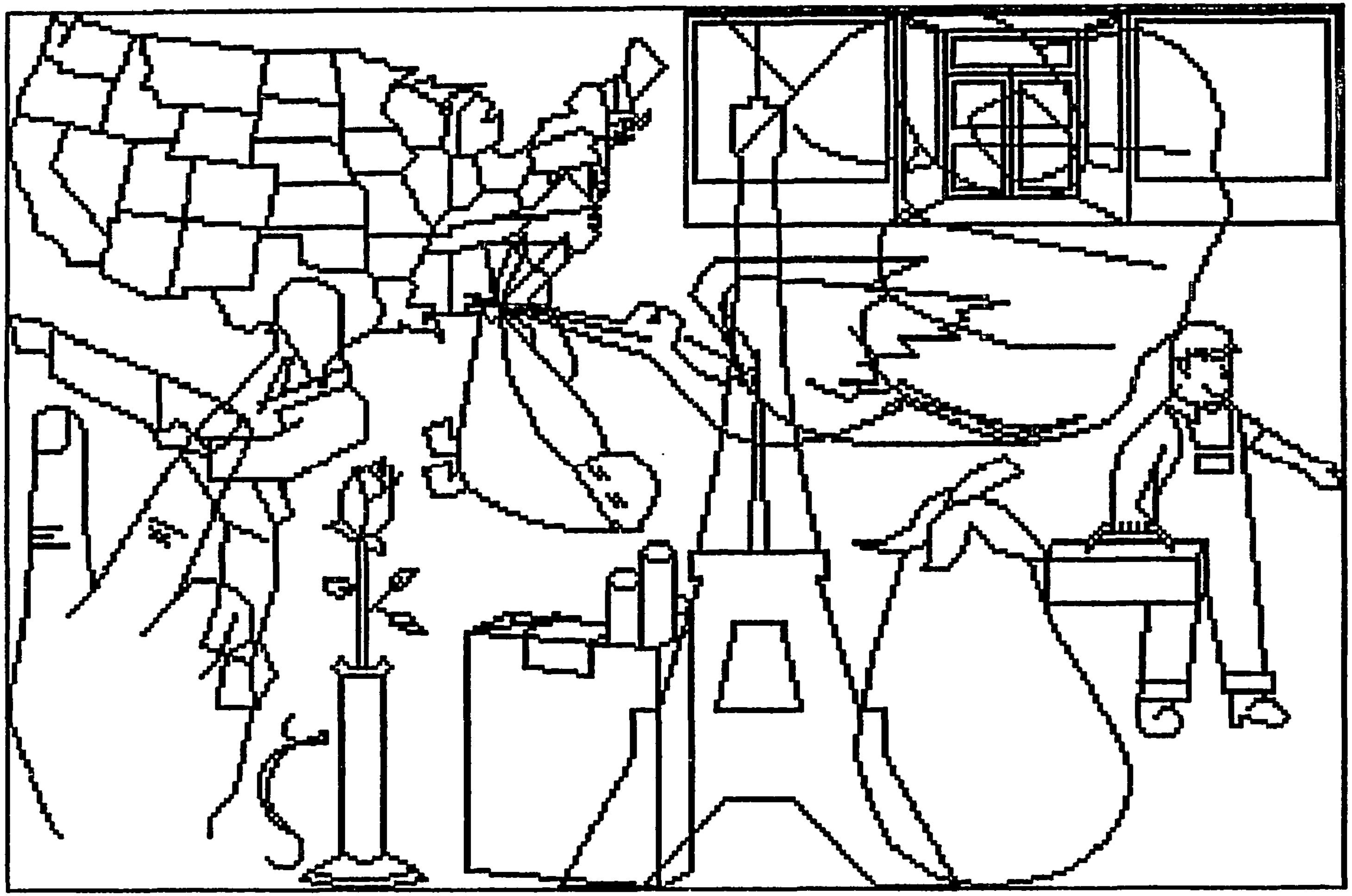


Figure 12.67. Image fig8

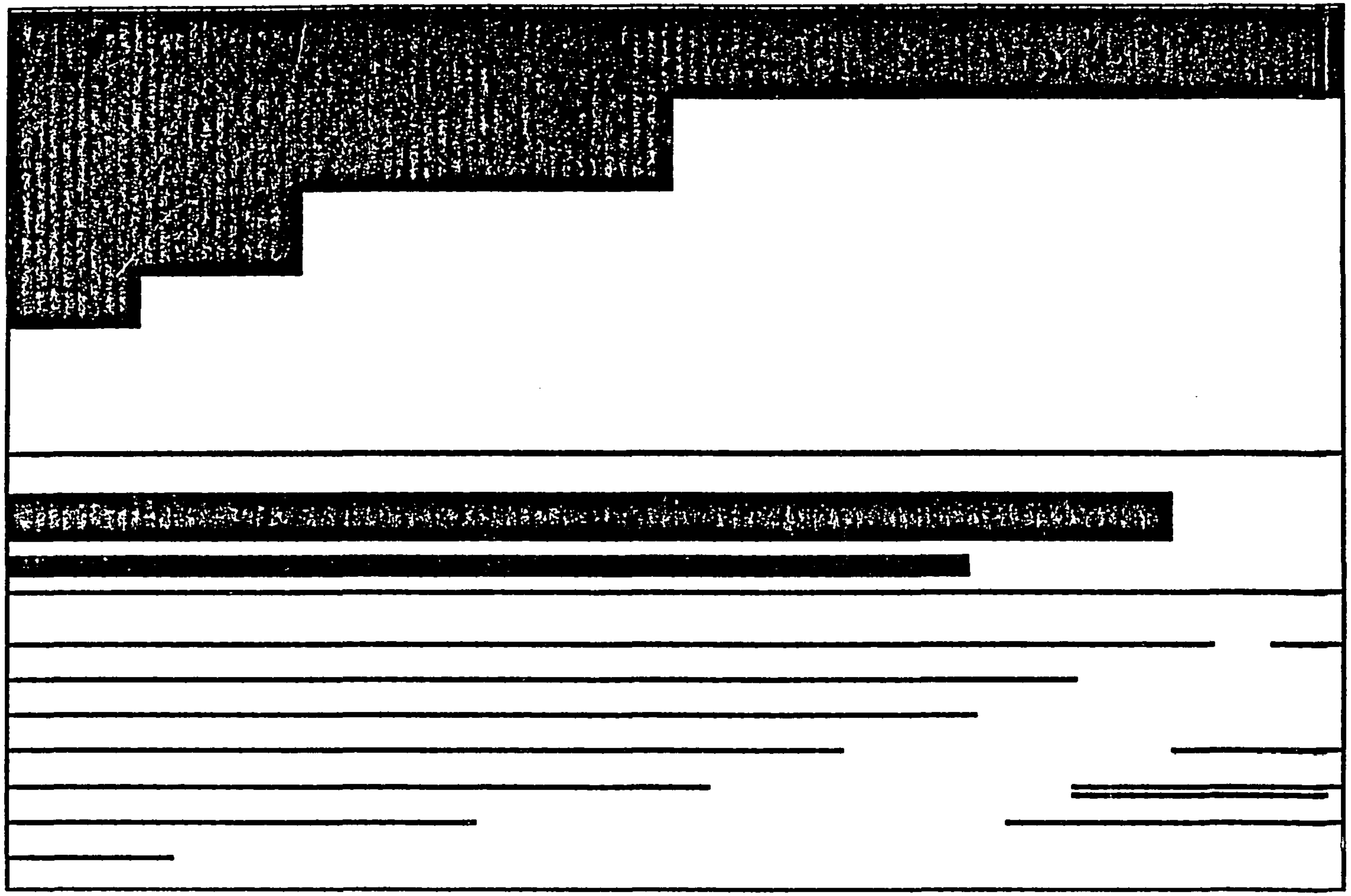


Figure 12.68. Image blok3

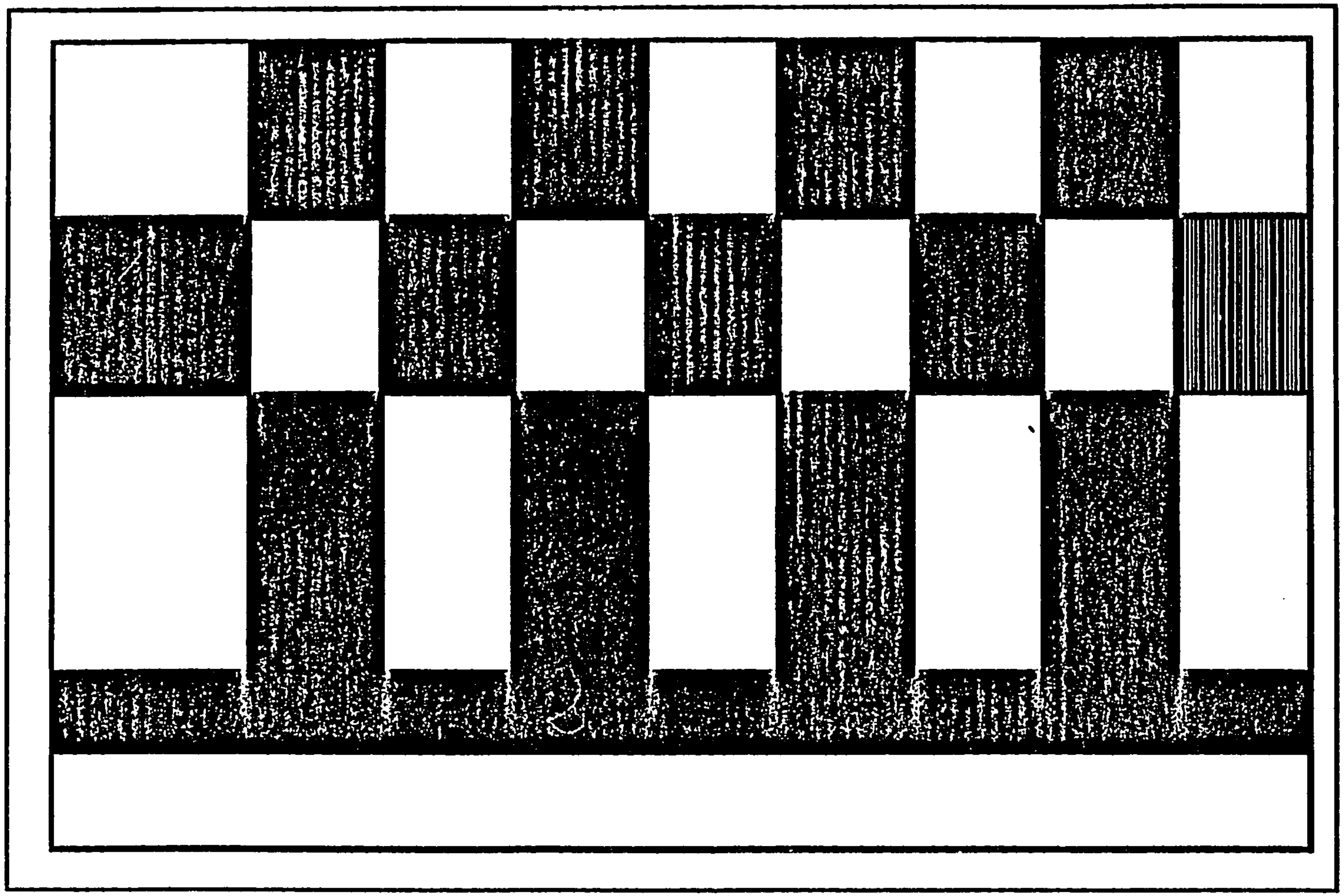


Figure 12.69. Image blok6

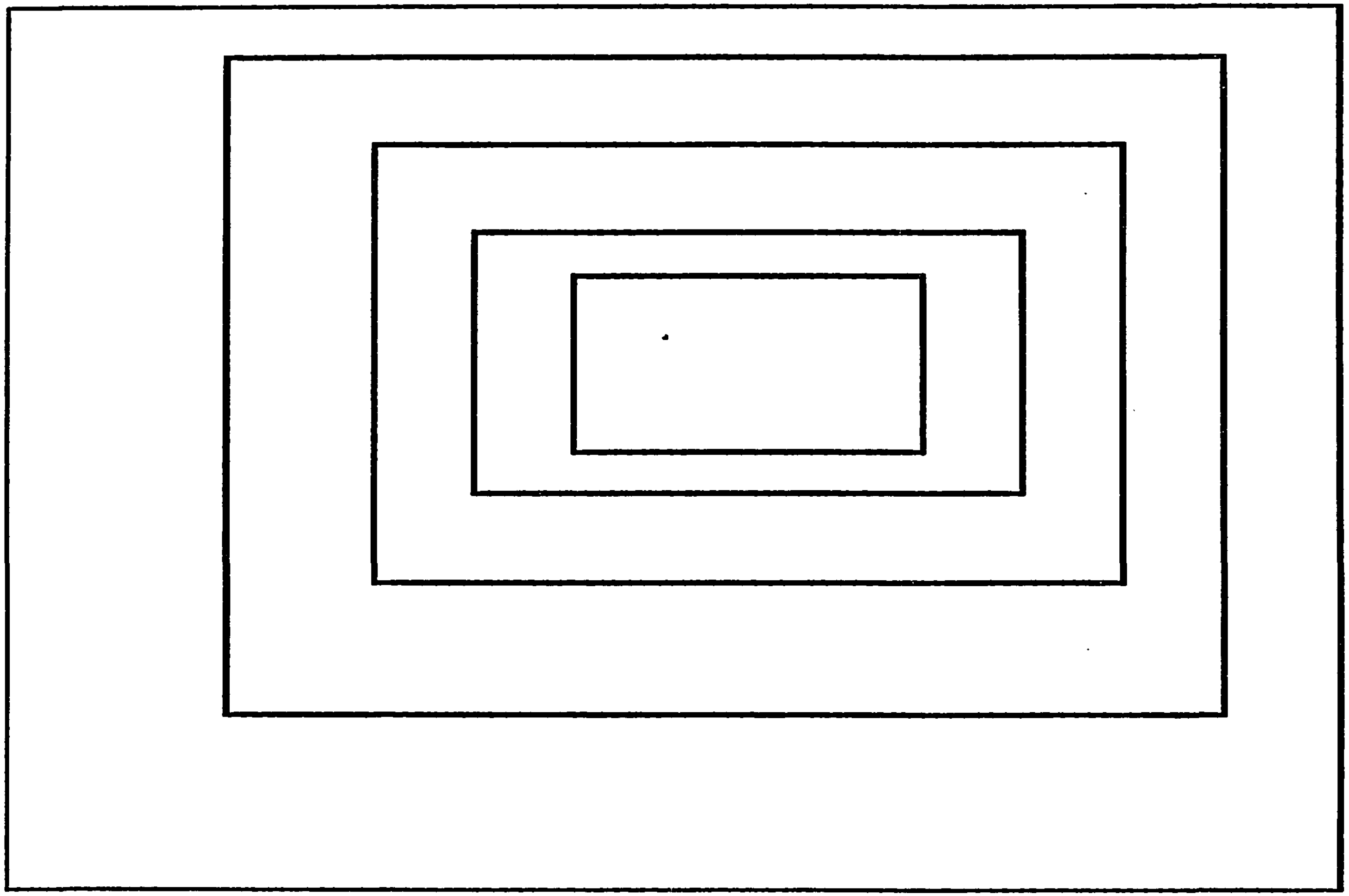


Figure 12.70. Image boxes

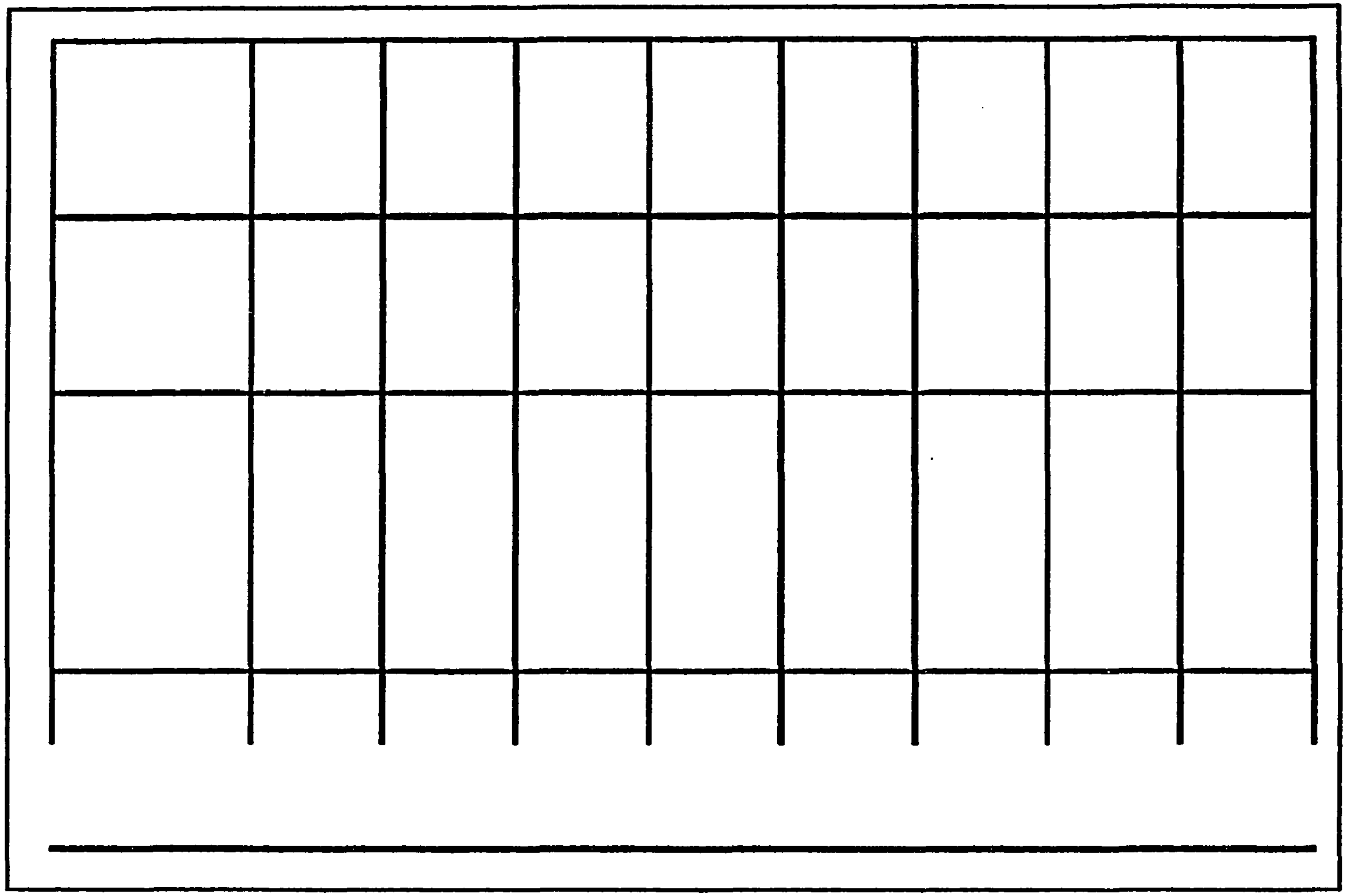


Figure 12.71. Image lines

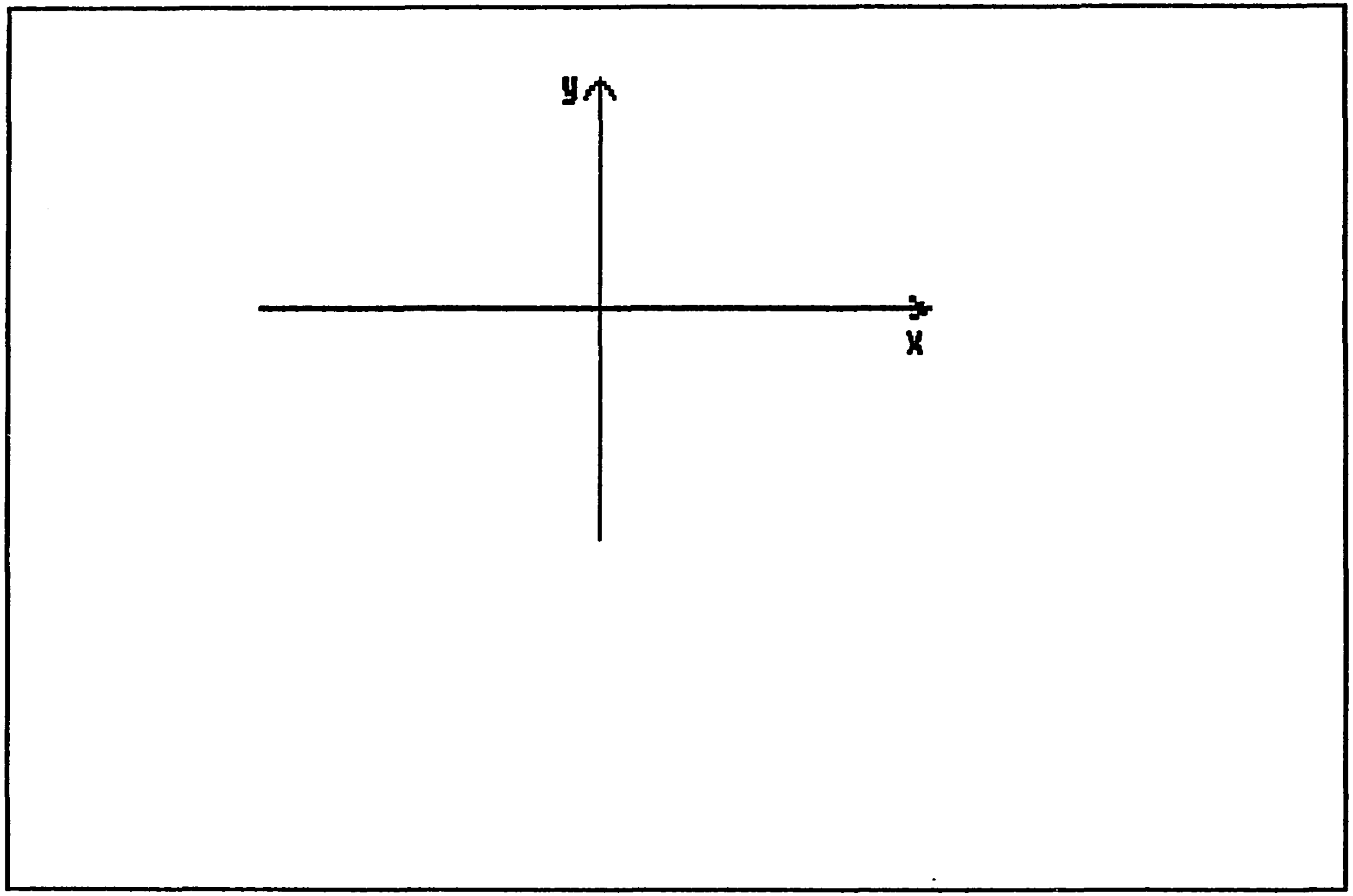


Figure 12.72. Image test1

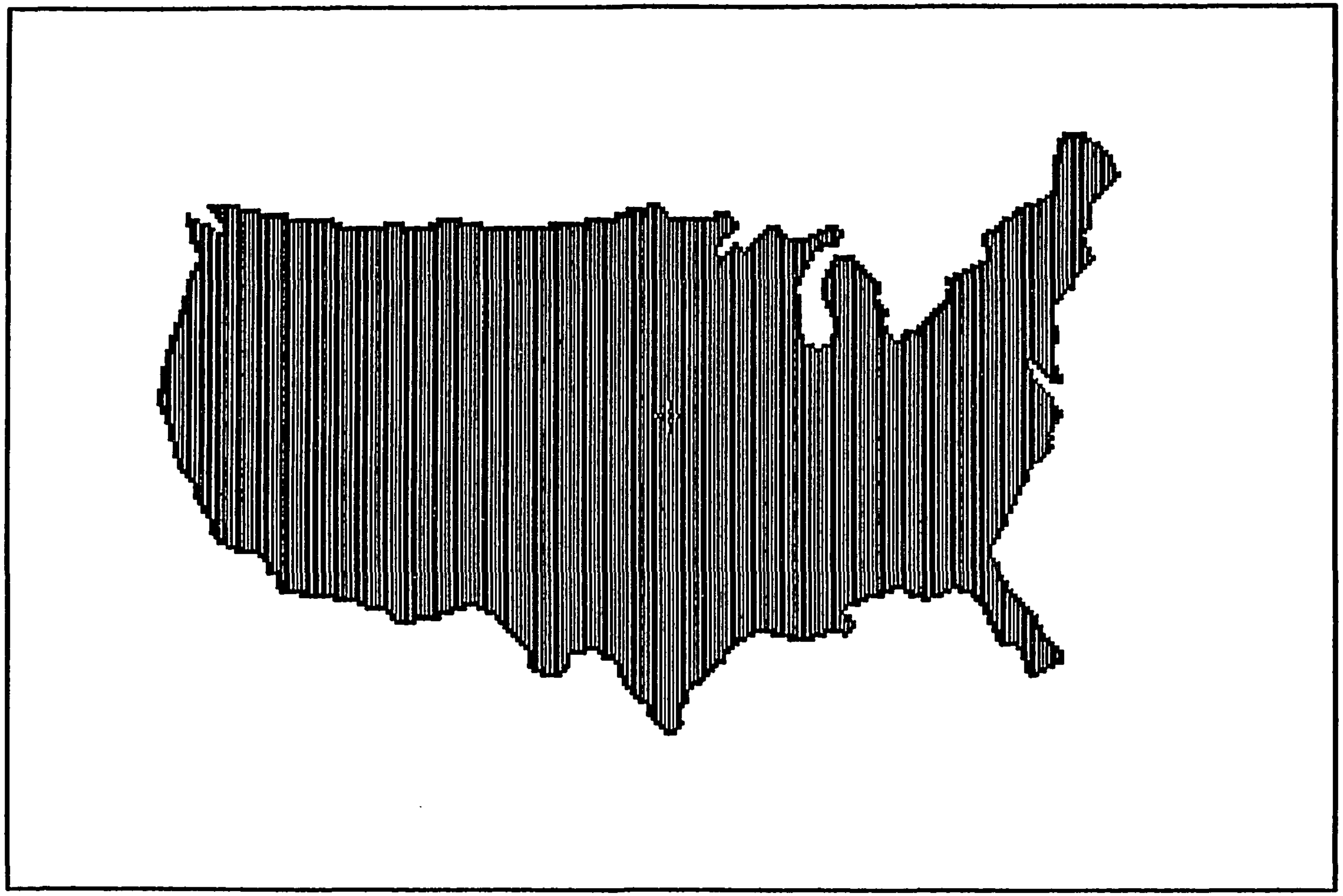


Figure 12.73. Image usamap

13. APPENDIX B. PROGRAM LIST OF THE CCITT ONE DIMENSIONAL COMPRESSION TECHNIQUE

The C programs in this appendix and the following appendices were compiled with Microsoft C Compiler version 4.0 and used the library functions of this compiler.

The assembly programs in this appendix and the following appendices were assembled with Microsoft Assembler version 4.0.

13.1. File Main.c

```
/* ==============================================================
 *  init_screen():   Function to initialize the screen, by setting
 *                   the mode and choosing the screen to display.
 *  get(x1,y1,x2,y2,buffer ): Takes the portion of the screen with
 *                            the x,y coordinates and saves it in
 *                            the buffer.
 * cmprs_line(uncmprsdbufr):  Apply  the  CCITT  one-Dimensional
 *                            compression technique, using a modified
 *                            Huffman table, to compress each line of
 *                            the specified portion of the screen and
 *                            put  the  result  in  the  uncompressed
 *                            buffer.
 * scrfilebufr: Array to hold the output of get(). The first 2 bytes
 *              hold the "xlength" of the block; the second two bytes
 *              hold  the  "ylength"  of  the  block.  The  size  of
 *              "scrfilebufr"  is  set  to  the  maximum  size of the
 *              blocks we want to capture.
 * xsize       : Horizontal length, in bits, of each line.
 * ysize       : Block height in bits.
 * tstart      : Time at start of compression or decompression.
 * tend        : Time at end of compression  or decompression.
 * cmprstime   : Compression time.
 * dcmprstime  : Decompression time.
 * ==============================================================
 */

#include        <stdio.h>
#include        <memory.h>
#include        <dos.h>
#include        <io.h>
#include        <fcntl.h>
#include        <malloc.h>
#define         LINT_ARGS
#define         screensize      16384
#define         XMAX            640
#define         YMAX            200
#define         HI_RES          6
#define         TEXT_MODE       3
#define         ulong           unsigned  long

void            get(int,int,int,int,char *);
```

```
unsigned        cmprs_line(char *);
void            dcmprs_line_ld();
unsigned        gttime();
void            print_results(char *, int, int, int, int,
                                 unsigned, unsigned, float);

static          float           avgfactor;
static          ulong           totalcmprsbits=0;
static          unsigned        cmprstime,dcmprstime;
                                   /* window coordinates.           */
static          int             x1,y1,x2,y2;
                                   /* figure input file.            */
                char            datafile[41];

main(argc,argv)
int     argc;
char    *argv[];

{
static          char            scrfilebufr[4+(XMAX/8)*(YMAX)];
static          unsigned        cmprsbufr[XMAX][(YMAX+32)/16];
static          char            *uncmprsbufr;
unsigned                        xsizeinbytes,xsize;
unsigned                        tstart,tend;
unsigned                        cmprsfactor[200];
register        unsigned        i,ysize;

                                   /* No data were entered at the   */
if( argc < 6 )                     /* command line.                 */
   {
        printf("enter x1 y1 x2 y2 \n");
        scanf("%d %d %d %d",&x1,&y1,&x2,&y2);
        while((getchar())!='\n')   /* Read the  end  of  the  line */
                                   /* marker.                       */
                ;
   }
else
   {
        x1=atoi(argv[2]); y1=atoi(argv[3]);
        x2=atoi(argv[4]); y2=atoi(argv[5]);
   }
if( argc > 1 )
        strcpy( datafile, argv[1] );
                                   /* Read data from the input file*/
init_screen(argc);                 /* and dump it to the screen.    */
uncmprsbufr= scrfilebufr;
uncmprsbufr+=4;                    /* Skip over "xsize" and "ysize"*/

                                   /* Get the specified portion of */
                                   /* the screen into "scrfilebufr"*/
```

```
get(x1,y1,x2,y2,(char *)scrfilebufr);
for(i=0;i<=55000;i++) ;            /*  A delay loop.               */
setscmode(TEXT_MODE);
ysize=y2-y1+1;
xsize=x2-x1+1;
xsizeinbytes= (xsize/8)+((xsize%8)>0) ;
                                   /* First  two  numbers  in  the */
                                   /* "screenfilebufr" represent   */
*(unsigned *)scrfilebufr=xsize;    /* the width and the height of  */
*(unsigned *)(scrfilebufr+2)=ysize;/* of the block.                */

printf("starting to compress \n");
tstart=gttime();                    /* Get the starting  time for  */
                                    /* the compression. Initialize */
                                    /* "comprsbufr" and the other  */
                                    /* static variables.           */
init_cmprsdblk((unsigned *)cmprsbufr);
init_line_parm(xsize);
for(i=1;i<=ysize;i++)
        {
        cmprsfactor[i]=cmprs_line(uncmprsbufr);
                                   /* Point to the next uncom-     */
        uncmprsbufr+=xsizeinbytes; /* pressed line on the screen.  */
        }
tend=gttime();                     /* Get the time at the end of   */
                                   /* the compression.             */
for(i=1;i<=ysize;i++)
        {
        totalcmprsbits=totalcmprsbits+cmprsfactor[i];
        cmprsfactor[i]=xsize/cmprsfactor[i];
        }
if(tend>tstart)
        cmprstime=tend-tstart;
else
        cmprstime=(6000-tstart)+tend;
printf("compression ended\n");

for(i=1;i<=ysize;i+=1)
        {                          /* Print  the  results  on  the */
                                   /* screen.                      */
        printf("%8u",cmprsfactor[i]);
        }
avgfactor=(float ) xsize * ysize/totalcmprsbits;

                                   /* Initialize "scrnfilebufr" to */
                                   /* ASCII zero.                  */
memset((scrfilebufr+4),'\0',16000);
printf(" starting to decompress \n");
tstart=gttime();                   /* Start of the decompression.  */
init_dcmprsbfr((scrfilebufr+4),xsize);
```

```
init_cmprs(cmprsbufr);
for(i=0;i<ysize;i++)
        dcmprs_line_ld();
tend=gttime();                     /* End the decompression.       */
if(tend>tstart)
        dcmprstime=tend-tstart;
else
        dcmprstime=(6000-tstart)+tend;
                                   /* If no argument was entered   */
                                   /* at the command line then     */
if( argc < 2 )                     /* display data to the screen.  */
   {
        setscmode(HI_RES);
        put(x1,y1,scrfilebufr);
        getchar();
        setscmode(TEXT_MODE);
   }
print_results(datafile,x1,y1,x2,y2,cmprstime,dcmprstime,avgfactor);
}
/*----------------------  END  main()  ------------------------------*/
/*----------------------  END  main.c  -------------------------------*/
```

13.2. File Cmprsln.c

```
/*
 *====================================================================
 * FUNCTIONS :
 *
 * cmprs_lastbits( word, no. of bits, color) : Compress the bits that
 *     did not fit into the word boundary (connect with the previous
 *     bits in the whole words portion of the line to be compressed.)
 * get_cmprs_reslt() : Returns the no. of compressed bits since the
 *     last time we zeroed "cmprscounter". This function is in the
 *     file "update.c", which in turn has the update() function that
 *     updates the compressed line after each compression.
 * init_lastbits (no. bits that did not fit into the line boundary) :
 *     Pass the number of the last bits to the file "clast.c".
 * swapbyts( from, to , number of words) : Swap the high and low byte
 *     of each word stored in 'from' and store the result in 'to'; do
 *     it for the passed number of words.
 *
 * VARIABLES :
 *
 * oldlineptr  : Pointer to the current line of uncompressed buffer.
 * newlineptr  : Pointer to the compressed line.
 * xsize       : Horizontal length, in bits, of each line.
 * currentword : Pointer to current position, in words ,
 *               in the "uncmprsbufr."
 * lastbits    : Number of bits in the last word of the uncompressed
 *               line if the number of bits in a line does not fit on
```

```
*               the word boundary.
* nmbrwords   : Word length of the portion of the uncompressed line
*               that fits in the word boundary.
* color       : Color of the current bit.
* lastcolor   : Color of the last bit processed in the whole words
*               portion of a line.
* bitcolor    : Color of the current bit (temporary storage.)
* word        : Current word in the uncompressed line.
* bitpos      : Index to the position in "word".
                bitpos = 16 for the left-most bit and
*                         1  for the right-most bit.
*=================================================================
*/

#include        <dos.h>
#define         LINT_ARGS
#define         BLACKBIT        0
#define         WHITEBIT        1
#define         ENDBITS         2

unsigned        get_cmprs_reslt();
void            init_lastbits(unsigned);
void            init_cmprsdblk(unsigned *);
void            update_cmprsdblk(unsigned,int);
void            cmprs_lastbits(unsigned,unsigned,int);
void            swapbyts(unsigned *,unsigned *,unsigned);

static unsigned         lastbits,nmbrwords;

/*===================== cmprs_line() =============================*/
unsigned        cmprs_line (oldlineptr)
char            *oldlineptr;

{
unsigned        *currentword;
int             wordcount;
int             color,lastcolor,bitcolor;
unsigned        bitcontr=0;
register        unsigned        word,bitpos;

wordcount=nmbrwords;            /* Initialize the variables.    */
currentword=(unsigned *)oldlineptr;
set_cmprscontr_to_zero();
swapbyts((unsigned *)oldlineptr,(unsigned *) oldlineptr,nmbrwords);
word=*currentword;
if ((word)&0x8000)              /* Is bit 16 in "word" white ?  */
        {                       /* Yes, bit 16 was white.       */
        update_cmprsdblk(0,BLACKBIT);
        color=WHITEBIT;
        }
```

```
else
        {                           /* Bit 16 was black.             */
        color=BLACKBIT;
                                    /* Negate the word so we can     */
        word=~word;                 /* check for the new color.      */
        }
                                    /* We assume "xsize" >= 16, to   */
                                    /* take care of "xsize" < 16. We*/
bitpos=16;                          /* have to modify the code here.*/
while(color<ENDBITS)                /* While not end of line, do.    */
        {
                                    /* While the color is the same   */
                                    /* and we are still inside       */
                                    /* "currentword", do.            */
        while( (word&0x8000) && (bitpos > 0) )
                {
                bitcontr++;
                bitpos--;           /* Bit position in a word.       */
                word=word<<1;       /* Get the next bit in bit 16.   */
                }
        if(bitpos > 0)              /* Still inside "currentword" ? */
                {
                update_cmprsdblk(bitcontr,color);
                word=~word;
                color=(color) ? 0 : 1;
                bitcontr=0;
                }
                                    /* Done with all the bits in     */
        else                        /* the current word.             */
                {
                bitpos=16;          /* Start again with bit 16       */
                currentword++;      /* of the next word.             */

                                    /* If the color is black then    */
                                    /* negate the word pointed to by*/
                                    /* "currentword" to check for    */
                                    /* the color later.              */
                word= (color) ? *currentword : ~(*currentword);
                                    /* Test for the end of the line */
                                    /* marker.                       */
                if(--wordcount == 0)
                        {           /* Save the last color in this   */
                                    /* line.                         */
                        lastcolor=color;
                                    /* Signal "eol" to the outer     */
                                    /* loop.                         */
                        color=ENDBITS;
                        }
                }
        }
```

```
if(lastbits == 0)                   /* Does the line fit in the word*/
                                    /* boundary ?                  */
        update_cmprsdblk(bitcontr,lastcolor);
else
        cmprs_lastbits(*currentword,bitcontr,lastcolor);
if(color>ENDBITS)
        printf(" ****** error in color, color=%d /n",color);
                                    /* Return the number of bits    */
return(get_cmprs_reslt());          /* in that compressed line.     */
}
/* ----------------------- END cmprs_line() -----------------------*/

/*===================== init_line_parm() ========================*/
/* Initialize some static variables to the appropriate values.     */
/*==============================================================*/
void init_line_parm(xsize)
unsigned        xsize;

{
nmbrwords=xsize/16;
lastbits=xsize & 0x000f;            /* Let "lastbits" = "xsize" % 16*/
init_lastbits(lastbits);
}
/* ---------------------- END init_line_param() ------------------- */
/* ------------------------- END cmprsln.c ------------------------ */
```

13.3. File Cupdt.c

```
/*
 *================================================================
 * STATIC VARIABLES :
 *
 * bitsleft      : Number of bits still vacant in the compressed word,
 *                 it starts with 16 bits left in the word.
 * cmprscounter : Count the number of bits in the compressed block
 *                 which is filled from left to right.
 * cmprsdwordptr: Pointer to the current word position in the
 *                 compressed block.
 *================================================================
 */

static  int             bitsleft;
static  unsigned        cmprscounter;
static  unsigned        *cmprsdwordptr;
/*===================== UPDATE_CMPRSDBLK() ======================*/
/* This is function update_cmprsdblk( bitcounter, color), where    */
```

```
/* bitcounter is the number of consecutive bits of current color.  */
/*================================================================*/
void    update_cmprsdblk(uncmprsdbitscont,color)
unsigned        uncmprsdbitscont;
register        int     color;

{
   struct       FAXDATA
                {
                                  /* Code for a sequence of bits  */
                                  /* of type color and run-length */
                                  /* = # of the uncompressed bits.*/
                        unsigned     bits;
                                  /* Length of  the code  in  the */
                                  /* bits.                        */
                        int          length;
                };

                                  /* Initialize "FAX". FAX[0][] ==*/
                                  /* black data , FAX[1][] ==     */
                                  /* white data.                  */
static  struct  FAXDATA FAX[2][74]={ {
        0x35,8, 0x7,6,  0x7,4,  0x8,4,  0xb,4,  0xc,4,  0xe,4,
        0xf,4,  0x13,5, 0x14,5, 0x7,5,  0x8,5,  0x8,6,  0x3,6,
        0x34,6, 0x35,6, 0x2a,6, 0x2b,6, 0x27,7, 0xc,7,  0x8,7,
        0x17,7, 0x3,7,  0x4,7,  0x28,7, 0x2b,7, 0x13,7, 0x24,7,
        0x18,7, 0x2,8,  0x3,8,  0x1a,8, 0x1b,8, 0x12,8, 0x13,8,
        0x14,8, 0x15,8, 0x16,8, 0x17,8, 0x28,8, 0x29,8, 0x2a,8,
        0x2b,8, 0x2c,8, 0x2d,8, 0x4,8,  0x5,8,  0xa,8,  0xb,8,
        0x52,8, 0x53,8, 0x54,8, 0x55,8, 0x24,8, 0x25,8, 0x58,8,
        0x59,8, 0x5a,8, 0x5b,8, 0x4a,8, 0x4b,8, 0x32,8, 0x33,8,
        0x34,8, 0x1b,5, 0x12,5, 0x17,6, 0x37,7, 0x36,8, 0x37,8,
        0x64,8, 0x65,8, 0x68,8, 0x67,8} , {

        0x37,10, 0x2,3,   0x3,2,   0x2,2,   0x3,3,   0x3,4,
        0x2,4,   0x3,5,   0x5,6,   0x4,6,   0x4,7,   0x5,7,
        0x7,7,   0x4,8,   0x7,8,   0x18,9,  0x17,10, 0x18,10,
        0x8,10,  0x67,11, 0x68,11, 0x6c,11, 0x37,11, 0x28,11,
        0x17,11, 0x18,11, 0xca,12, 0xcb,12, 0xcc,12, 0xcd,12,
        0x68,12, 0x69,12, 0x6a,12, 0x6b,12, 0xd2,12, 0xd3,12,
        0xd4,12, 0xd5,12, 0xd6,12, 0xd7,12, 0x6c,12, 0x6d,12,
        0xda,12, 0xdb,12, 0x54,12, 0x55,12, 0x56,12, 0x57,12,
        0x64,12, 0x65,12, 0x52,12, 0x53,12, 0x24,12, 0x37,12,
        0x38,12, 0x27,12, 0x28,12, 0x58,12, 0x59,12, 0x2b,12,
        0x2c,12, 0x5a,12, 0x66,12, 0x67,12, 0xf,10,  0xc8,12,
        0xc9,12, 0x5b,12, 0x33,12, 0x34,12, 0x35,12, 0x6c,13,
        0x6d,13, 0x4a,13} } ;

register unsigned code;           /* Code for the run of the pels.*/
int               length;         /* Length of the above code     */
```

```
unsigned          multiple;         /*  = "uncmprsdbitscont" / 64.  */
unsigned          bitcont;          /* Local run-length.            */

                                    /* To get the least significant */
static  unsigned  mask1=0x003f;     /* 6 bits.                      */

                                    /* Is "uncmprsdbitscont" a      */
                                    /* multiple of 64 ?             */
if((multiple=(uncmprsdbitscont>>6))>0)
        {
                                    /* Compress the multiple of     */
        bitcont=multiple+63;        /* 64 part.                     */
        code=FAX[color][bitcont].bits;
        length=FAX[color][bitcont].length;
        cmprscounter=cmprscounter+length;
                                    /* Is old "bitsleft" > length ? */
        if ((bitsleft=bitsleft-length)>0)
                                    /* Put  the  new code  at  the  */
                                    /* current  compressed  word,   */
                                    /* using the new "bitsleft" to  */
                                    /* put  it  in  the  correct    */
                                    /* position.                    */
                (*cmprsdwordptr)|=code<<(bitsleft);
        else                        /* The old "bitsleft" <= length.*/

                {                   /* Negate "bitsleft" and put the*/
                                    /* part of the code that fills  */
                                    /* the word in the compressed   */
                                    /* word.                        */
                (*cmprsdwordptr)|=(code) >> (-bitsleft);
                                    /* Move to a new  word  and put */
                                    /* the rest of the code in a    */
                                    /* new compressed word, filling */
                                    /* from the left to the right.  */
                *(++cmprsdwordptr)=(code) <<
                                      (bitsleft = (16 + bitsleft));
                }
                                    /* Now compress the part that   */
                                    /* is less than 64 bits.        */
                                    /* If the no. of bits = 640 we  */
        if(multiple<10)             /* skip putting the zero part.  */
                {
                                    /* "bitcont" is the remainder of*/
                                    /* dividing "uncmprsdbitscont"  */
                                    /* by 64.                       */
                bitcont=uncmprsdbitscont & mask1;
                                    /* Get the corresponding  code  */
                                    /* and the "code-length".       */
                code=FAX[color][bitcont].bits;
                length=FAX[color][bitcont].length;
```

```
                                    /* Update "cmprscounter" by the*/
                                    /* "code-length".              */
                cmprscounter=cmprscounter+length;
                                   /* If there are still more      */
                                   /* unprocessed bits in the      */
                                   /* current word then put the    */
                                   /* compressed bits in the       */
                                   /* corresponding part of the    */
                                   /* word in the compressed buffer*/
                if(( bitsleft=bitsleft-length)>0 )
                        (*cmprsdwordptr)|=code<<(bitsleft);
                else
                        {          /* Otherwise split the code     */
                                   /* among the current and next   */
                                   /* words of the compressed      */
                                   /* buffer.                      */
                        ((*cmprsdwordptr))|=(code) >> (-bitsleft);
                        (*++cmprsdwordptr)=(code) <<
                                     (bitsleft = (16 + bitsleft));
                        }
                }
        }
                                   /* Run-length was less than     */
else                               /* 64 bits.                     */
        {
                                   /* Get the corresponding number */
                                   /* of bits and "run-length"     */
                                   /* then update "cmprscounter".  */
        code=FAX[color][uncmprsdbitscont].bits;
        length=FAX[color][uncmprsdbitscont].length;
        cmprscounter=cmprscounter+length;
                                   /* Same case as the one before. */
        if ((bitsleft=bitsleft-length)>0)
                (*cmprsdwordptr)|=code<<(bitsleft);
        else
                {
                ((*cmprsdwordptr))|=(code) >> (-bitsleft);
                (*++cmprsdwordptr)=(code) <<
                                 (bitsleft = (16 + bitsleft));
                }
        }
}
/*---------------------- UPDATE_CMPRSDBLK() -----------------------*/

/*===================== INIT_CMPRSDBLK() ========================*/
/* Initialize the compression buffer pointer to the first word of  */
/* the space allocated, set the compression counter to zero and    */
/* start with the most left bit of the first word in the compressed*/
/* buffer.                                                         */
/*=================================================================*/
```

```
void    init_cmprsdblk(newblkptr)
unsigned        *newblkptr;

{
        cmprsdwordptr=newblkptr;
        bitsleft=16;
        cmprscounter=0;
}
/*--------------------- END INIT_CMPRSDBLK() ----------------------*/

/*==================== get_cmprs_reslt() ==========================*/
/* This function returns the number of compressed bits since last  */
/* initialization of "cmprscounter".                                */
/*================================================================*/
unsigned        get_cmprs_reslt()

{
return(cmprscounter);
}
/*----------------- END get_cmprs_reslt() -------------------------*/

/*================= set_cmprscontr_to_zero() =====================*/
/* Set_cmprscontr_to_zero() :it sets "cmprscounter" to zero. Use it*/
/* if you are compressing a block and want to get "cmprscounter"   */
/* for each line alone.                                            */
/*================================================================*/
void    set_cmprscontr_to_zero()
{
cmprscounter=0;
}
/*----------------- END set_cmprscontr_to_zero() ------------------*/
/* ------------------------ END cupdt.c -------------------------- */
```

13.4. File Clast.c

```
#include        <dos.h>
#define         LINT_ARGS
#define         BLACKBIT        0
#define         WHITEBIT        1
#define         ENDBITS         2
#define         flip(word)      \
                {                               \
                        inregs.x.ax=word;       \
                        inregs.h.bl=inregs.h.al;        \
                        inregs.h.al=inregs.h.ah;        \
                        inregs.h.ah=inregs.h.bl;        \
                        word=inregs.x.ax;               \
                }
void            update_cmprsdblk( unsigned, int);
```

```
static          unsigned        lastbits;

/*============================ CMPRS_LASTBITS =====================*/
/* The bits left in the  last  word after  compressing the  whole   */
/* screen should  be  handled as a  special  case.  First the word  */
/* should be flipped, or swapped.  It would  not be  necessary to   */
/* check for the word  boundary since we are  sure that the number  */
/* of bits left is less than 16.                                    */
/*==================================================================*/
cmprs_lastbits(word,bitcontr,color)
register        unsigned        word;           /* Last word.         */
unsigned        bitcontr;                       /* Counter of bits left */
int             color;                          /* Last color.        */

{
struct bits
        {
        unsigned        rest    :15;
        unsigned        bit16   :1;
        } ;
union
        {
        struct  bits    b;
        unsigned        w;
        } wordbitsl;
union   REGS    inregs;
int             bitcolor;
register        int     bitpos;

        flip(word)
        bitpos=0;
        while(color < ENDBITS)
        {
                wordbitsl.w=word;  /* Last word.                      */
                                   /* Loop  until  either  "color"    */
                                   /* changes  or  all  bits  are     */
                                   /* processed.                      */
                while( (wordbitsl.b.bit16 == color) &&
                        (bitpos < lastbits) )
                {
                        bitcontr++;
                        bitpos++;
                                    /*  Get  the  next  bit.          */
                        wordbitsl.w = word = word << 1;
                }
                if(bitpos < lastbits)
                {                  /* The  color  changed,  hence     */
                                   /* update the compressed buffer.*/
                        update_cmprsdblk(bitcontr,color);
```

```
                                   /* Let "color" = new color.      */
                        color=wordbits1.b.bit16;
                                   /* Start looking for a new run. */
                        bitcontr=0;
                }
                else
                {                  /* All  bits  were  processed,  */
                                   /* update the compressed buffer */
                                   /* and  exit  the  main  loop.  */
                        update_cmprsdblk(bitcontr,color);
                        color=ENDBITS;
                }
        }
}
/*--------------------- END cpmrs_lastbits() ----------------------*/

/*====================== init_lastbits() ==========================*/
/* Initialize "lastbits" to the no. of bits in last word of the    */
/* uncompressed line.                                              */
/*=================================================================*/
void    init_lastbits(lastcont)
unsigned        lastcont;

{
lastbits=lastcont;
}
/*------------------- END init_lastbits() -------------------------*/
/* ---------------------- END clast.c ---------------------------- */
```

13.5. File Dcmprsln.c

```
#include        <stdio.h>
#include        <io.h>
#include        "colordef.h"

int     update_cmprs(int);
int     uncmprs_blak(),                 uncmprs_white();
int     match_blak(int *,int *),        match_white(int *,int *);
int     update_dcmprs_blakmk(int),      update_dcmprs_whitemk(int);
int     update_dcmprs_blakreg(int),     update_dcmprs_whitereg(int);

/*======================== DCMPRSLN() =============================*/
/* This function decompresses or decodes one horizontal line using */
/* the  CCITT  one-dimensional  coding  standard.  The  function   */
/* consists of a while loop to process all the codes in a line.    */
/*=================================================================*/
void    dcmprs_line_1d()
{
                                   /* Each line is assumed to begin*/
```

```
                                   /* with a black run, if it does */
                                   /* not, then the code of zero   */
                                   /* black run was inserted before*/
                                   /* the compressed code of the   */
                                   /* line at the compression time.*/
                                   /* Decode the compressed buffer */
                                   /* until the end of line is     */
                                   /* encountered.                 */
        while(  uncmprs_blak() && uncmprs_white() )
                                ;
}
/*----------------------- END DCMPRSLN() --------------------------*/

/*====================== UNCMPRS_BLAK() ==========================*/
/* When either a make-up or a terminating black code is processed, */
/* both of the compressed and decompressed buffer are updated. The */
/* latter is updated by sending the corresponding number of bits   */
/* to that buffer.                                                 */
/*=================================================================*/
uncmprs_blak()

{
int                             clrbits,codebits;
register        int             *clrbitsptr=&clrbits;
register        int             *codebitsptr=&codebits;

match_blak(clrbitsptr,codebitsptr);
                                   /* In  case   "clrbit"   is     */
                                   /* smaller  than 0   then a     */
                                   /* make-up  code was  encount-  */
                                   /* ered  as  a first  code, so  */
                                   /* updated   compression  and   */
                                   /* decompression  buffers.      */
if(*clrbitsptr<0)
        {
        *clrbitsptr=-*clrbitsptr;
        update_cmprs(*codebitsptr);
        update_dcmprs_blakmk(*clrbitsptr);
                                   /* Find new clrbits & codebits  */
        match_blak(clrbitsptr,codebitsptr);
        }
                                   /* Update "cmprsbufr" with  the */
                                   /* first  terminating  code     */
                                   /* length  encountered.         */
update_cmprs(*codebitsptr);
                                   /* Put  "clrbits"  black  pels  */
                                   /* in the decompression buffer. */
                                   /* If the line ended return 1   */
                                   /* else return 0.               */
return( update_dcmprs_blakreg(*clrbitsptr));
```

```
}
/*---------------------- END UNCMPRS_BLK() ------------------------*/

/*===================== UNCMPRS_WHITE() ==========================*/
/* When either a make-up or a terminating white code is processed, */
/* both of compression and decompression buffer are updated. The   */
/* latter is updated by sending the corresponding number of bits   */
/* to that buffer.                                                 */
/*=================================================================*/
uncmprs_white()
{

int                             clrbits,codebits;
register        int             *clrbitsptr=&clrbits;
register        int             *codebitsptr=&codebits;

match_white(clrbitsptr,codebitsptr);
                                   /* Refer  to  the  comments in  */
if(*clrbitsptr<0)                  /* function  uncmprs_blak.      */
        {
        *clrbitsptr=-*clrbitsptr;
        update_cmprs(*codebitsptr);
        update_dcmprs_whitemk(*clrbitsptr);
        match_white(clrbitsptr,codebitsptr);
        }
update_cmprs(*codebitsptr);
return( update_dcmprs_whitereg(*clrbitsptr));
}
/*--------------------- END UNCMPRS_WHITE() -----------------------*/
/* --------------------- END dcmprsln.c ------------------------ */
```

13.6. File Dupdtc.c

```
/*
 *=================================================================
 * STATIC VARIABLES :
 *
 * cbitsremain : Bits remained in a given word, initial value is
 *              16 bits.
 * currentword : Holds the current word to be decoded.
 * nextwordptr : Points to the next word to be processed after the
 *              current word.
 * nextword    : It is set to the contents of word pointed to by
 *              "nextwordptr". After each code match, "nextword" is
 *              masked so that it will contain the unused portion,
 *              it is right justified.
 *              The rest of it is filled with zeros.
 * rightbitsword : Masks to get 1st bit, 1st and 2nd bits and so on.
 * leftbitsword : Masks to get 16th bit, 16th and 15th bits and so on.
```

```
 *===================================================================
 */

static  unsigned        currentword;
static  unsigned        nextword,*nextwordptr;
static  unsigned        cbitsremain;
static  unsigned        rightbitsword[]={0,0x0001,0x0003,0x0007,
                                          0x000f,0x001f,0x003f,
                                          0x007f,0x00ff,0x01ff,
                                          0x03ff,0x07ff,0x0fff,
                                          0x1fff,0x3fff,0x7fff,
                                          0xffff};
unsigned                leftbitsword []={0,0x8000,0xc000,0xe000,
                                          0xf000,0xf800,0xfc00,
                                          0xfe00,0xff00,0xff80,
                                          0xffc0,0xffe0,0xfff0,
                                          0xfff8,0xfffc,0xfffe,
                                          0xffff};

/*======================= UPDATE_CMPRS() ========================*/
/* This function updates "currentword", which is a window into the */
/* compressed buffer.                                               */
/*==================================================================*/
update_cmprs(codelngth)
int     codelngth;

{                                 /* Variable "tempword" is not    */
                                  /* necessary, it is used to     */
                                  /* speed processing.            */
register        unsigned        tempword;
register        int             difference;

tempword = currentword;
tempword <<= codelngth;           /* Get  rid  of  this  code.     */

                                  /* Can  the  vacant  place  in   */
                                  /* "currentword" be filled from */
                                  /* what is left in nextword?    */
if((difference = cbitsremain-codelngth) > 0)
        {                         /* Yes,  "bitsremain"  is  big   */
                                  /* enough.                      */

                                  /* Copy the new bits of the code*/
                                  /* into the places vacant due to*/
                                  /* the mathed code.             */
        tempword |= nextword>>(difference);
        }
else
        {                         /* No, the code  bits remaining */
                                  /* in "nextword" can't fill the */
```

```
                                   /* places  vacated  due  to the */
                                   /* matched  code.               */

                                   /* Correct "difference".        */
        difference =- difference;
                                   /* Copy  all the  code bits in  */
                                   /* "nextword" to their correct  */
                                   /* positions  in  "tempword".   */
        tempword |= nextword << (difference);
                                   /* Advance "nextwordptr" and    */
                                   /* copy its content to          */
                                   /* "nextword".                . */
        nextword = *(++nextwordptr);
                                   /* Adjust "difference" then use */
                                   /* it  to  copy the  necessary  */
                                   /* part from the new "nextword" */
                                   /* into "tempword".             */
        tempword |= nextword >> (difference=(16- (difference)) );
        }
                                   /* Mask the used part to zeros. */
nextword &= rightbitsword[difference];
cbitsremain = difference;          /* Update  "cbitsremain".       */
currentword = tempword;            /* Update "currentword".        */
}
/*-------------------- END UPDATE_CMPRS ---------------------------*/

/*======================= init_cmprs =============================*/
init_cmprs(cmprsbfrptr)
unsigned        *cmprsbfrptr;

{
cbitsremain = 16;
currentword = *(cmprsbfrptr);
nextword    = *(nextwordptr=cmprsbfrptr+1);
}
/*--------------------- End init_cmprs ----------------------------*/

/*====================== MATCH_BLAK ==============================*/
/* It looks at the content of "currentword"(currentword is a window*/
/* that  slides on the "cmprsdbfr") from left to right ( up to bit */
/* 9 ) and tries to match the first four bits with a code of black */
/* runs whose length is four bits. If no match is found it tries   */
/* to match the first 5 bits and so on until it finds a match. The */
/* last bits to be looked at are the first 8 bits. It is assumed   */
/* that a match should be found otherwise an error message is sent */
/* to the screen and the program is halted.                        */
/* It returns the length of the  matched  code and the length  of  */
/* the corresponding run in locations pointed to by "codebitsptr"  */
/* and"clrbitsptr" respectively.                                   */
/*================================================================*/
```

```
match_blak(clrbitsptr,codebitsptr)
register        int     *clrbitsptr;
int                     *codebitsptr;

{
                                   /* Huffman table for  the black */
                                   /* codes.   It  is  read  from  */
                                   /* right  to  left  with  the   */
                                   /* vacant  bits  filled  with   */
                                   /* zeros in every word.         */
static unsigned         BLK_CODES[] =
                        {
                                   /* BARRAY_4 bits.               */
                            0x7000,0x8000,0xb000,0xc000,0xe000,
                            0xf000,
                                   /* BARRAY_5 bits.               */
                            0x9800,0xa000,0x3800,0x4000,0xd800,
                            0x9000,
                                   /* BARRAY_6 bits.               */
                            0x1c00,0x2000,0x0c00,0xd000,0xd400,
                            0xa800,0xac00,0x5c00,
                                   /* BARRAY_7 bits.               */
                            0x4e00,0x1800,0x1000,0x2e00,0x0600,
                            0x0800,0x5000,0x5600,0x2600,0x4800,
                            0x3000,0x6e00,
                                   /* BARRAY_8 bits.               */
                            0x3500,0x0200,0x0300,0x1a00,0x1b00,
                            0x1200,0x1300,0x1400,0x1500,0x1600,
                            0x1700,0x2800,0x2900,0x2a00,0x2b00,
                            0x2c00,0x2d00,0x0400,0x0500,0x0a00,
                            0x0b00,0x5200,0x5300,0x5400,0x5500,
                            0x2400,0x2500,0x5800,0x5900,0x5a00,
                            0x5b00,0x4a00,0x4b00,0x3200,0x3300,
                            0x3400,0x3600,0x3700,0x6400,0x6500,
                            0x6800,0x6700
                        };
                                   /* Run-lengths  corresponding   */
                                   /* to the codes in "BLK_CODES". */
                                   /* Make-up runs are stored  as  */
                                   /* negative    values     to    */
                                   /* distinguish   them   from    */
                                   /* the terminating runs.        */
static int      BLK_RUNS[] =
                        {          /*  BCODE_4 bits.               */
                            2  ,3  ,4  ,5   ,6   ,7   ,
                                   /*  BCODE_5 bits.               */
                            8   ,9   ,10   ,11   ,-64   ,-128 ,
                                   /*  BCODE_6 bits.               */
                            1, 12, 13, 14, 15, 16, 17, -192  ,
                                   /*  BCODE_7 bits.               */
```

```
                                        18, 19, 20, 21, 22, 23, 24, 25, 26,
                                        27, 28, -256,
                                           /*  BCODE_8 bits.           */
                                        0, 29, 30, 31, 32, 33, 34, 35, 36,
                                        37 , 38, 39, 40, 41, 42, 43, 44, 45,
                                        46, 47, 48, 49, 50, 51, 52, 53, 54,
                                        55, 56, 57, 58, 59, 60, 61, 62, 63,
                                        -320, -384, -448, -512, -576, 640
                                 };
                                           /* The black codes are grouped  */
                                           /* in the "BLK_CODES" array     */
                                           /* according to their length.   */
                                           /* Their corresponding runs are */
                                           /* stored in "BLK_RUNS" array.  */
                                           /* The  first element in        */
                                           /* "BGROUPS" is equal to the no.*/
                                           /* of the pairs. First no. in   */
                                           /* each  pair  is the length of */
                                           /* the code in bits. Second no. */
                                           /* is the number of codes with  */
                                           /* this length.
static  int  BGROUPS[]={5,   4,6,  5,6,  6,8,  7,12,  8,42 };
register        word;

word = currentword;
switch (1)
   {
        case 1:
        {                          /* Find the first part of "word"*/
                                   /* that can be matched to a code*/
                                   /* of a black run. When a match */
                                   /* occurs return the "clrbits"  */
                                   /* and "codebits".              */
        if( match_all_bits(word,BLK_CODES,BLK_RUNS,BGROUPS,
                                clrbitsptr,codebitsptr) )
                break;
        }
        default : {
                  printf("Wrong code encountered in 'match_blak'\n");
                  exit(0) ;
                  }
     }
}
/*---------------------- END MATCH_BLAK -------------------------*/

/*===================== MATCH_WHITE ===========================*/
/* Codes of length = 2, 3, 4, 5, 6, 7, 8, 9 are  processed  in  a  */
/* tree data structure in order to find a match for them with the  */
/* first 2, 3,...9 left bits of "currentword". Whenever a match is */
/* found we exit from the tree.  If no match is found in the tree, */
```

```
/* the function  looks at the content of currentword (current word */
/* is a window that slides  on the "cmprsdbfr") from left to right */
/* (up to bit 4) and tries to match the first ten bits with a code */
/* of white runs whose length is ten  bits. If no match is found   */
/* it tries to match the first 11 bits and so on until it finds  a */
/* match. The last bits to be looked at are the first 13 bits. It  */
/* is  assumed  that a  match should be found otherwise an error   */
/* message is sent to the screen the and program is halted.        */
/* The function returns the length of the  matched  code and  the  */
/* length of the corresponding run  in locations  pointed  to  by  */
/* "codebitsptr" and   "clrbitsptr" respectively.                  */
/*================================================================*/
match_white(clrbitsptr,codebitsptr)
int             *clrbitsptr,*codebitsptr;

{
                                   /* See comment for "BLK_CODES". */
static  unsigned        WHITE_CODES[] =
                        {
                                   /* Codebits = 10.               */
                                0x05c0, 0x0600, 0x0200, 0x03c0,
                                0x0dc0,
                                   /* WARRAY_11 bits.              */
                                0x0ce0, 0x0d00, 0x0d80, 0x06e0,
                                0x0500, 0x02e0, 0x0300,
                                   /* WARRAY_12 bits.              */
                                0x0ca0, 0x0cb0, 0x0cc0, 0x0cd0,
                                0x0680, 0x0690, 0x06a0, 0x06b0,
                                0x0d20, 0x0d30, 0x0d50, 0x0d60,
                                0x0d70, 0x06c0, 0x06d0, 0x0da0,
                                0x0db0, 0x0540, 0x0550, 0x0560,
                                0x0570, 0x0640, 0x0650, 0x0520,
                                0x0530, 0x0240, 0x0370, 0x0380,
                                0x0270, 0x0280, 0x0580, 0x0590,
                                0x02b0, 0x02c0, 0x05a0, 0x0660,
                                0x0670, 0x0c80, 0x0c90, 0x05b0,
                                0x0330, 0x0340, 0x0350,
                                   /* WARRAY_13 bits.              */
                                0x0360, 0x0368, 0x0250
                        };
                                   /* See comment for "BLK_RUNS".  */
static  int     WHITE_RUNS[] =
                        {
                                   /* WCODE_10 BITS.               */
                                16, 17, 18, -64, 0,
                                   /* WCODE_11 bits.               */
                                19, 20, 21, 22, 23, 24, 25,
                                   /* WCODE_12 bits.               */
                                26, 27, 28, 29, 30, 31, 32, 33,
                                34, 35, 36, 37, 38, 39, 40, 41,
```

```
                                42, 43, 44, 45, 46, 47, 48, 49,
                                50, 51, 52, 53, 54, 55, 56, 57,
                                58, 59, 60, 61, 62, 63, -128, -192,
                                -256, -320, -384, -448,
                                   /* WCODE_13 bits.              */
                                -512, -576, 640
                        };
static  unsigned  WGROUPS[]={4, 10,5,   11,7,   12,44,  13,3 };
register        unsigned        tmpword,word;

word = currentword;
switch (1)
   {
        case 1:
        {
        if(word & 0x8000)               /*  Bit 16 = 1.             */
                {
                if(word & 0x4000)       /* Bit 15 = 1 hence code=2.*/
                        *clrbitsptr = 2;
                else                    /*     Bit 15 = 0.          */
                        *clrbitsptr = 3;
                *codebitsptr = 2;       /*  Code length = 2.        */
                break;
                }
        if(word & 0x4000)               /*    Bit 15 = 1.           */
                {
                if(word & 0x2000)       /*    Bit 14 = 1.           */
                        *clrbitsptr=4;  /*    Code = 4 .            */
                else                    /*    Bit 14 = 0.           */
                        *clrbitsptr=1;  /*    Code = 4.             */
                *codebitsptr=3;         /*    Code length = 3.      */
                break;
                }
        if(word & 0x2000)               /*    Bit 14 = 1.           */
                {
                if(word & 0x1000)       /*    Bit 13 = 1.           */
                        *clrbitsptr=5;  /*    Code = 5.             */
                else                    /*    Bit 13 = 0.           */
                        *clrbitsptr=6;  /*    Code = 6.             */
                *codebitsptr=4;         /*    Code length = 4.      */
                break;
                }
        if(word & 0x1000)               /*    Bit 13 = 1.           */
                {
                if(word & 0x0800)       /*    Bit 12 = 1.           */
                        {
                        *clrbitsptr=7;  /*    Code = 7.             */
                        *codebitsptr=5; /*    Code length = 5.      */
                        break;
                        }
```

```
                else                    /*    Bit 12 = 0.          */
                        {               /*    Bit 11 = 1.          */
                        if(word & 0x0400)
                                        /*    Code = 8.            */
                                *clrbitsptr=8;
                        else            /*    Bit 11 = 0.          */
                                        /*    Code = 9.            */
                                *clrbitsptr=9;
                        *codebitsptr=6; /*    Code length = 6.     */
                        break;
                        }
                }
                                   /* By reaching this points it   */
                                   /* means that only 4 zero bits  */
                                   /* were found.                  */
                                   /*------------------------------*/
                                   /* mask with 1111 1110 00...    */
                                   /* to handle runs 10, 11, 12    */
        if((tmpword=(word&0xfe00)) == 0x0800)
                { *codebitsptr=7;       *clrbitsptr=10; break;  }
        if(tmpword==0x0a00)
                { *codebitsptr=7;       *clrbitsptr=11; break;  }
        if(tmpword==0x0e00)
                { *codebitsptr=7;       *clrbitsptr=12; break;  }
                                   /*------------------------------*/
                                   /* mask with 1111 1111 00...    */
                                   /* to handle runs 13, 14, 15    */
        if((tmpword=(word&0xff00)) == 0x0400)
                { *codebitsptr=8;       *clrbitsptr=13; break;  }
        if(tmpword==0x0700)
                { *codebitsptr=8;       *clrbitsptr=14; break;  }
        if((word&0xff80)==0x0c00)
                { *clrbitsptr=15;       *codebitsptr=9; break;  }
                                   /*------------------------------*/
                                   /* find  the first  part of     */
                                   /* "word" that can be matched   */
                                   /* to a code of a white run.    */
                                   /* When a match occurs return   */
                                   /* the "clrbits" and "codebits".*/
        if( match_all_bits(word,WHITE_CODES,WHITE_RUNS,WGROUPS,
                                clrbitsptr,codebitsptr) )
                break;
        }
        default : {
                  printf(
                    " Wrong code encountered in 'match_white'\n");
                  exit(0) ;
                  }
   }
}
```

```
/*--------------------- END MATCH_WHITE() ------------------------*/
/* ---------------------- END dupdtc.c ------------------------- */
```

13.7. File Dupdtd.c

```
/*
 *==================================================================
 * STATIC VARIABLES :
 *
 * dbitsremain : Bits remaining in a given byte, initial value is
 *               8 bits.
 * xsize       : Horizontal dimension of the block = length of
 *               each line.
 * xlength     : Counter for number of bits processed in the
 *               current line.
 * linestart   : Points to the start byte of every line in
 *               compressed buffer.
 * currentbyteptr : Points to the current byte, in the decompression
 *                  buffer, to be filled.
 * currentbyte    : Equals the contents of byte pointed to by
 *                  "currentbyteptr".
 * leftbitsbyte   : An array of masks to get the 16th bit, the 16th
 *                  and 15th bits, and so on.
 * rightbitsbyte  : An array of masks to get the 1st bit, the 1st
 *                  and 2nd bits, and so on.
 *==================================================================
 */

#include        <memory.h>
#define         uchar   unsigned        char

static  int     dbitsremain;
static  int     xsize, xlength;
static  char    *linestart;
static  uchar   *currentbyteptr, currentbyte;
static  uchar   rightbitsbyte[]=
                     {0,0x01,0x03,0x07,0x0f,0x1f,0x3f,0x7f,0xff};
static  uchar   leftbitsbyte[] =
                     {0,0x80,0xc0,0xe0,0xf0,0xf8,0xfc,0xfe,0xff};

/*================== update_dcmprs_whitereg ====================*/
/* Put into the decompression buffer "dcmprsbufr" the exact number */
/* of white bits that equals the passed run length.               */
/*===============================================================*/
update_dcmprs_whitereg(clrbits)
                                  /* Number of white bits to be    */
register        int     clrbits;  /* added to the buffer.          */

{
```

```
register        int     difference;
unsigned                nmbrbytes;

difference = clrbits-dbitsremain;
if( clrbits >= (dbitsremain+8) )   /* Can we use memset() ?        */
        {                          /* YES we can, hence set the    */
                                   /* remaining bits of the current*/
                                   /* byte to 1's.                 */
        *currentbyteptr |= rightbitsbyte[dbitsremain];
                                   /* Divide by 8 to get the number*/
        nmbrbytes=(difference)>>3; /* of bytes that need to be     */
                                   /* updated.                     */
                                   /* Set "nmbrbytes" bytes to ones*/
        memset(++currentbyteptr,0xff,nmbrbytes);
        currentbyteptr +=nmbrbytes; /* Advance the pointer position*/
                                   /* If the  difference  was  not */
                                   /* divisible  by 8  then there  */
                                   /* are  some  bits  to be  set  */
                                   /* to ones in the next byte.    */
        if((difference=difference &0x7) !=0)
                *currentbyteptr=leftbitsbyte[(difference)];
        dbitsremain=8-(difference);
        }
else                               /* No we can not use memset().  */
        {
        if(difference < 0)
                {                  /* Only few bits need to be set */
                                   /* to one  within  the current  */
                                   /* byte, hence OR contents of   */
                                   /* "currentbyteptr" with the    */
                                   /* mask that is shifted left by */
                                   /* the negated difference.      */
                *(currentbyteptr) |= ( rightbitsbyte[clrbits] <<
                                            (dbitsremain-clrbits) );
                dbitsremain -=clrbits;
                }
        else                       /* There are some bits in the   */
                                   /* current and next byte to be  */
                                   /* set to one .                 */
                {
                                   /* Set  those  bits  left  in   */
                                   /* the current byte to 1's      */
                *currentbyteptr |=rightbitsbyte[dbitsremain];
                                   /* Set the required bits of the */
                                   /* next byte to one.            */
                *(++currentbyteptr) =leftbitsbyte[difference];
                dbitsremain = 8 - (difference);
                }
        }
                                   /* If the end of the line is    */
```

```
                                   /* reached then initialize the  */
if( (xlength+=clrbits) >= xsize )  /* variables to process the next*/
                                   /* line.                        */
        {
        xlength=0;
                                   /* If "dbitsremin"=8 this means */
                                   /* that  "currentbyteptr"  is   */
                                   /* pointing to the first byte of*/
                                   /* the next line and, of course,*/
                                   /* "dbitsremain" is correct.    */
                                   /* So start a new line.         */
        if(dbitsremain!=8)
                {
                dbitsremain=8;
                ++currentbyteptr;
                }
        linestart=currentbyteptr;
        return(0);
        }
else
        return(1);                 /* Line did not end yet.        */
}
/*------------------ END UPDATE_DCMPRS_WHITEREG() -----------------*/

/*==================== update_dcmprs_blakreg ======================*/
/* Put into the decompression buffer "dcmprsbufr" the exact number */
/* to black bits that equals the passed run length. Since initially*/
/* every bit in the buffer is set to zero, it is enough to advance */
/* the pointer by the run length.                                  */
/*=================================================================*/
update_dcmprs_blakreg(clrbits)
register        int     clrbits;

{
register        int     difference;
unsigned                nmbrbytes;

difference=clrbits-dbitsremain;
if(clrbits >= (dbitsremain+8) )    /* Update more than two bytes.  */
        {
                                   /* No need to set the remaining */
                                   /* bits of  the  current  byte  */
                                   /* to 0's  since the buffer is  */
                                   /* initialized to zero's.       */

                                   /* Divide by 8 to get the number*/
        nmbrbytes=(difference)>>3; /* of bytes to be updated.      */
                                   /* Advance the pointer position.*/
        currentbyteptr +=nmbrbytes+1;
                                   /* By ANDING  "difference"  with*/
```

```
                                   /* 0000 0111  we  get  the  bit */
                                   /* position  to  start  with  in*/
                                   /* the next process.            */
          dbitsremain=8-(difference &0x7 );
          }
else                               /* Update one or two bytes.     */
          {
          if(difference<0)
                                   /* Only  few  bits  need to  be */
                                   /* set  to  zero  within  the   */
                                   /* current byte, hence  advance */
                                   /* "dbitsremain" by "clrbits",  */
                                   /* thus bits = run-length are   */
                                   /* set to zero in the current   */
                                   /* byte.                        */
                    dbitsremain -=clrbits;
          else
                    {
                                   /* Advance  "dbitsremain"  by   */
                                   /* "clrbits", thus bits =  run- */
                                   /* length  are  set  to zero in */
                    ++currentbyteptr; /* the current and next byte.   */
                    dbitsremain=8- (difference);
                    }
          }
                                   /* If the end of line is reached*/
                                   /* then initialize the variables*/
if( (xlength+=clrbits) >= xsize )  /* to process the next line.    */
          {
          xlength=0;
                                   /* If "dbitsremin"=8 this means */
                                   /* that  "currentbyteptr"  is   */
                                   /* pointing  to  first  byte of */
                                   /* the next line and, of course,*/
                                   /* "dbitsremain" is correct.    */
          if(dbitsremain!=8)       /* So start a new line.         */
                    {
                    dbitsremain=8;
                    ++currentbyteptr;
                    }
          linestart=currentbyteptr;
          return(0);
          }
else
          return(1);               /* Line did not end yet.        */
}
/*------------------ END UPDATE_DCMPRS_BLAKREG() ------------------*/

/*=================== update_dcmprs_whitemk =====================*/
update_dcmprs_whitemk(clrbits)
```

```
int     clrbits;

{
register        int             difference;
register        unsigned        nmbrbytes;

                                /* Refer to the comments in the    */
                                /* function update_dcmprs_whitereg.*/
difference=clrbits-dbitsremain;
*currentbyteptr |= rightbitsbyte[dbitsremain];
nmbrbytes=(difference)>>3;
memset(++currentbyteptr,0xff,nmbrbytes);
currentbyteptr +=nmbrbytes;
if((difference=difference &0x7) !=0)
        *currentbyteptr=leftbitsbyte[(difference)];

dbitsremain=8-(difference);
xlength +=clrbits;
return(1);
}
/*------------------ END UPDATE_DCMPRS_WHITEMK() ----------------- */

/*==================== update_dcmprs_blakmk =======================*/
update_dcmprs_blakmk(clrbits)
register        int     clrbits;

{
register        int     difference;
unsigned        nmbrbytes;

                                /* Refer to comments in function   */
                                /* update_dcmprs_blakreg.          */
difference=clrbits-dbitsremain;
nmbrbytes=(difference)>>3;
currentbyteptr +=nmbrbytes+1;
dbitsremain=8-(difference &0x7);
xlength +=clrbits;
return(1);
}
/*------------------- END UPDATE_DCMPRS_BLAKMK() ------------------*/

                                   /* Even number of bytes only.   */
                                   /* If the line length is odd do */
static  unsigned  bytelinelngth;   /* not process the last byte.   */

/*======================== init_dcmprsbfr ========================*/
init_dcmprsbfr(dcmprsbfrptr,sizexbits)
unsigned          char  *dcmprsbfrptr;
int                     sizexbits;
```

```
{
linestart=currentbyteptr=dcmprsbfrptr;
bytelinelngth= ( ( ((xsize=sizexbits)/8) /2)*2);
xlength=0;
dbitsremain=8;
}
/*---------------------- END INIT_DCMPRSBFR() ---------------------*/

/*======================== adjst_line ===========================*/
/* swap every pair of bytes in every word of the current line.    */
/*==============================================================*/
adjst_line()
{
swapbyts(linestart,linestart,bytelinelngth);
}
/*----------------------- END ADJST_LINE() -----------------------*/
/* ----------------------- END dupdtd.c ----------------------- */
```

13.8. File Initscrn.c

```
#include        <stdio.h>
#include        <memory.h>
#include        <dos.h>
#include        <io.h>
#include        <fcntl.h>
#include        <malloc.h>
#define         LINT_ARGS
#define         SCREENSIZE      16384
#define         HI_RES          6
#define         TEXT_MODE       3
                                   /* window coordinates.          */
extern          int             x1,y1,x2,y2;
                                   /* figure input file.           */
extern          char            datafile[];

/* ====================== INIT_SCREEN ======================= */
/* init_screen(value) : Function to initialize the whole screen.   */
/* It takes its input interactively. If "value" is equal to one    */
/* then the input file was entered at the command line.            */
/* ========================================================== */
init_screen(value)
int     value;

{
char    *screenbufr;               /* Temporary buffer.            */
int     fhl,bytesread,modeval,loop=1;
char    flag,c;
                                   /* "src" is a far pointer       */
char    far     *src;              /* initialized to "screenbufr". */
```

```
if( value <= 1 )
{
        while(loop)
        {
                printf("enter name of data file \n");
                gets(datafile);
                printf("your data file is %s \n",datafile);
                printf("Are the values entered correct ?\n");
                printf("enter Y or N    ");
                flag=getchar();
                while( ((flag!='y')&&(flag!='n')) )
                {
                        flag=getchar();
                        printf("enter y or n ");
                        flag=getchar();
                }
                while((c=getchar()) !='\n')
                        ;
                if(flag=='y')
                        loop=0;
        }
}
setscmode(HI_RES);
                           /* do the first bank (even) by  */
                           /* allocating the half total    */
                           /* size.                        */
screenbufr=malloc(SCREENSIZE/2);
fhl = open(datafile,O_RDONLY|O_BINARY);
                           /* read the first bank.         */
bytesread=read(fhl,screenbufr,SCREENSIZE/2);
src=(char far *)(screenbufr+7);
                  /* The screen format has the first byte */
                  /* of the 1st bank at offset 8000 of the*/
                  /* screen segment. Move the data from   */
                  /* the file to that segment. Note that  */
                  /* in the screen segment the bytes      */
                  /* starting at offset 8000 till (8192-7)*/
                  /* will be filled with whatever the file*/
                  /* has. This part is not from the       */
                  /* physical screen.
movedata(FP_SEG(src),FP_OFF(src),0xb800,0x0000,
                                       (SCREENSIZE/2)-7);
bytesread=read(fhl,screenbufr,SCREENSIZE/2);
src=(char far *)(screenbufr);
                  /* The 1st seven bytes of the 2nd half  */
                  /* of the file are a continuation of the*/
                  /* (192-7) bytes that BASIC took from   */
                  /* the screen memory and dumped it to   */
                  /* the file. So the second half of the  */
```

```
                           /* screen starts after 7 bytes of the    */
                           /* 2nd part of the file. By copying the  */
                           /* second half of the file into offset   */
                           /* (0x2000-7) we will fill the 7 bytes   */
                           /* at (0x2000-7) then the 2nd half of    */
                           /* the screen will be copied to offset   */
                           /* (0x2000). This fills the odd part of  */
                           /* the screen. The remaining (192-7) of  */
                           /* the file will fill offset             */
                           /* (0x2000+8000) till offset             */
                           /* (0x2000+8000+(192-7)).                */
        movedata(FP_SEG(src),FP_OFF(src),0xb800,(0x2000-7),
                                                 SCREENSIZE/2);
        close(fhl);
        free(screenbufr);
}
/*-------------------------- END INIT_SCREEN ----------------------*/

/*========================== SETSCMODE ============================*/
/* Sets the screen to the desired video mode.                       */
/*================================================================*/
int setscmode(mode)                /* set the video mode function. */
int mode;

{
union REGS inregs;
union REGS outregs;
                                   /* return the code and the       */
                                   /* interrupt for function        */
                                   /* "gdosint".                    */
int ret_code,int_no;
                                   /* "set video mode" BIOS         */
                                   /* function call.                */
   inregs.h.ah=0;
   inregs.h.al=mode;
   ret_code = int86(0x10,&inregs,&outregs);
                                   /* return the code to check for */
                                   /* any errors.                   */
   return(ret_code);
}
/*---------------------- END SETSCMODE ----------------------------*/
/* --------------------- END initscrn.c ------------------------- */
```

13.9. File Gttime.c

```
#include        <dos.h>
#define         LINT_ARGS
#define         INT_TIME        0x1a
```

```
/*======================== GTTIME ==============================*/
/* It returns the current time, only the seconds and the hundredths*/
/* of a second. The return value is the addition  of  the  two, in */
/* hundredths of a second.                                        */
/*==============================================================*/
unsigned        gttime()

{
union REGS inregs;
union REGS outregs;
unsigned        tc;

inregs.h.ah=0x2c;
intdos(&inregs,&outregs);
tc = (outregs.h.dl) + (100 * outregs.h.dh);
return(tc);
}
/*------------------------ END GTTIME () ------------------------*/
/* ----------------------- END gttime.c ------------------------ */
```

13.10 File Print.c

```
#include        <io.h>
#include        <stdio.h>

/*==================== PRINT_RESULTS() =========================*/
/* Print the results to the output file. The data to be printed out*/
/* are the compression time, the decompression time and the       */
/* compression factor.                                            */
/*==============================================================*/
print_results(thefile,x1,y1,x2,y2,cmprstime,dcmprstime,avgfactor)
char            thefile[41];
unsigned        x1,y1,x2,y2;
unsigned        cmprstime,dcmprstime;
float           avgfactor;
{
FILE            *outfile;

printf(" Compression factor is %f \n", avgfactor) ;
printf(
  "Compression   time is %u  in 1/100 of a second \n", cmprstime );
printf(
  "Decompression time is %u  in 1/100 of a second \n", dcmprstime );
                                   /* Send data to table.dat file. */
if( (outfile = fopen( "table.dat", "r" )) == NULL )
{
                                   /* Open the file for writing.   */
                                   /* Print the table heading too. */
   outfile = fopen( "table.dat", "w" ) ;
```

```
      fprintf(outfile,
         "File name            x1    y1    x2    y2    cmprs     cmprs   ");
      fprintf(outfile,"dcprs \n" );
      fprintf(outfile,
         "                                             fctor     time    ");
      fprintf(outfile,"time  \n" );
      fprintf(outfile,
         "----------------------------------------------------------");
      fprintf(outfile,"----------------------------\n");
           }
   else
           {                           /* Appending.                   */
           outfile = fopen( "table.dat", "a" ) ;
           }
                                       /* Output formats.              */
   fprintf(outfile,"%-20s  %3u  %3u  %3u  %3u  %6.2f    %4u  %5u\n",
           thefile, x1, y1, x2, y2, avgfactor, cmprstime, dcmprstime );
   fclose(outfile);
   }
   /*---------------------- END PRINT_RESULTS() ----------------------*/
   /* ------------------------ END print.c ------------------------ */
```

13.11. File Geth.asm

```
NAME            GET
TITLE           GET GRAPHIC SCREEN  GETH

_TEXT   SEGMENT  BYTE PUBLIC 'CODE'
_TEXT   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP

_DATA   SEGMENT  WORD PUBLIC 'DATA'
EXTRN           __CHKSTK:NEAR
MASK2           DB      0FFH
                DB      080H,0C0H,0E0H,0F0H,0F8H,0FCH,0FEH
GTSETMOD        DB      0
_DATA   ENDS
PUBLIC          _get,MASK2
                                        ; INPUTS :
BUFFER          EQU     [BP+12]         ; POINTER TO MEMORY BUFFER.
Y0              EQU     [BP+6]          ; Y OF UPPER LEFT CORNER.
Y2              EQU     [BP+10]         ; Y OF LOWER RIGHT CORNER.
```

```
INXO            EQU     [BP+4]
INX2            EQU     [BP+8]
                                        ; WORK VARIABLES :
                                        ; X OF THE BYTE IN WHICH IS
                                        ; THE UPPER LEFT CORNER OF
                                        ; THE BLOCK
X0              EQU     [BP-2]          ; 0 <= X0 <= 79 BYTES.
                                        ; SOURCE INCREMENT AFTER
                                        ; EACH LINE MOVE. DEST_INC =
                                        ; DX, IT ISN'T DEFINED HERE
                                        ; BUT WE NEED IT TO SKIP
SORC_INC        EQU     [BP-4]          ; LINES OF THE OTHER BANK.
                                        ; SOURCE ( SCREEN ) OFFSET
SORC_INDX2      EQU     [BP-6]          ; OF THE FIRST BYTE.
                                        ; DESTINATION (BUFFER) OFFSET
DEST_INDX2      EQU     [BP-8]          ; OF THE FIRST BYTE IN THE
SHFT_RGT        EQU     [BP-10]         ; 2ND BANK. SEE FINDPARAM
LINE_CNTR       EQU     [BP-12]         ; NO. OF LINES IN EACH BANK.
PXLENGTH        EQU     [BP-14]

ODDORG = 02000H

_BSS            SEGMENT word    public  'BSS'
                EVEN
                                ; "LAST_MASK" IS A VARIABLE TO BE
                                ; INITIALIZED FROM THE VALUES IN
                                ; "MASK1". "LAST_MASK" IS USED IN
                                ; "BLOKX" AND "XBLOK". IT IS OF BYTE
                                ; SIZE. IN FUNCTION PUT() "BLOKX"
                                ; WILL HAVE "RIGHT_MASK" OF SIZE
                                ; BYTE AND "XBLOK" WILL HAVE
                                ; "LAST_MASK" OF SIZE WORD, SO IT IS
LAST_MASK       DB      ?       ; DIFFERENT FROM THIS "LAST_MASK".
                EVEN
LT1             DB      ?
RT1             DB      ?
BIT1            DW      ?
_BSS            ENDS

_GET    PROC    NEAR
        PUSH    BP
        MOV     BP,SP
        MOV     AX,14
        CALL    __chkstk
        PUSH    DI
        PUSH    SI
        PUSH    ES
        PUSH    DS
        PUSH    DX
        PUSH    CX
```

```
         PUSH   BX
         PUSH   AX
GET_SMODE:
         MOV    AH,15
         INT    16
         CMP    AL,4
         JNE    HIGH_RES
         MOV    RT1,2
         MOV    LT1,1
         MOV    WORD PTR BIT1,3
         JMP    FIND_PARAMS
HIGH_RES:
         CMP    AL,6
         JNE    NOT_GRAPH
         MOV    RT1,3
         MOV    WORD PTR BIT1,7
         MOV    LT1,0
         JMP    FIND_PARAMS
NOT_GRAPH:
         JMP    GT_DONE
FIND_PARAMS:
         CALL   FINDPARAM
COMMENT *
FIND_PARAMS WILL RETURN "DX = XLENGTH" (CASE1 AND CASE3_B) OR
"XLENGTH-1" (FOR CASE 2 AND CASE3_A) WHERE "XLENGTH"= NO. OF BYTES
NEEDED TO STORE EACH LINE.
BX = COUNTER FOR Y LINES= NO. OF LINES IN THE FIRST BANK.
CX = KIND OF BLOCK.
*
INIT_BUFFER:
         MOV    AX,DS              ; LET ES = DS.
         MOV    ES,AX
         MOV    DI,BUFFER          ; DI = ADDRESS OF THE 1ST BYTE IN
                                   ; THE BUFFER. STORE "XLENGTH" IN THE
         MOV    AX,PXLENGTH        ; 1ST WORD OF THE BUFFER.
         STOSW                     ; "XLENGTH" IS IN PELS.
         MOV    AX,Y2
         SUB    AX,Y0
         INC    AX                 ; AX = Y2-Y0 + 1.
                                   ; STORE "YLENGTH" IN THE 2ND WORD OF
         STOSW                     ; THE BUFFER.
MOVE_SETUP:
         MOV    AX,80
         SUB    AX,DX
         MOV    SORC_INC,AX        ; "SORC_INC" = 80 - SIZE.
         MOV    AX,Y0
         SHR    AX,1               ; AX = NUMBER OF THE 1ST LINE ON THE
                                   ; SCREEN.
         MOV    SI,DX              ; STORE "DX" IN "SI".
         MOV    BL,80
```

```
         MUL     BL
         MOV     DX,SI           ; RESTORE "DX".
         ADD     AX,X0           ; AX = (Y0/2) * 80 + X0.
         MOV     BX,Y0
         AND     BX,1            ; IF BL = 1 THEN Y0 IS ODD.
         JZ      ORG_EVEN        ; IF BL = 0 THEN Y0 IS EVEN.
                                 ; Y0 IS ODD SO ADD THE ORIGIN OFFSET
         ADD     AX,ODDORG       ; OF THE ODD BANK INTO THE SCREEN
                                 ; SEGMENT.
         MOV     BX,80
ORG_EVEN:
         MOV     SI,AX           ; SI = SOURCE INDEX OF THE FIRST BANK
         ADD     AX,BX           ; IF Y0 IS ODD THEN SORC_INDX2 =
         MOV     SORC_INDX2,AX   ; SOURCE INDEX1 + 80.
         MOV     AX,BUFFER       ; DEST_INDX1 WAS ALREADY INITIALIZED
         ADD     AX,4            ; TO 4 AFTER WE FILLED THE FIRST TWO
         ADD     AX,DX           ; WORDS OF THE BUFFER.
         MOV     DEST_INDX2,AX   ; DEST_INDX2 = BUFFER + 4 (DUE FIRST
                                 ; TWO WORDS) + DX (DUE TO THE FIRST
                                 ; LINE)
         MOV     AX,Y2
         SUB     AX,Y0
         INC     AX
                                 ; BX = AX = Y2-Y0 + 1 = "YLENGTH"
         MOV     BX,AX           ; IN PELS.
         SHR     AX,1            ; AX = NO. OF LINES IN THE SECOND
                                 ; BANK.
         MOV     LINE_CNTR,AX    ; SO STORE IT IN THE LINE COUNTER.
         AND     BX,1            ; IF "YLENGTH" IN PELS WAS ODD THEN
         ADD     BX,AX           ; BX = 1, HENCE ADD IT TO THE LINE
                                 ; COUNTER.
                                 ; STORE THE RESULT IN BX, WHICH WE
                                 ; USE AS Y LINES COUNTER.
         CMP     BX,0            ; IF BX (i.e. NO. OF Y LINES IN THE
         JBE     Y_ERROR         ; FIRST BANK) <= 0 THEN Y VALUES
                                 ; WERE WRONG)
         MOV     AX,0B800H       ; LET DS = B800 = SCREEN SEGMENT.
         MOV     DS,AX
         CLD                     ; JUST TO MAKE SURE.
CHOOSE:  JCXZ    ALL_OK
         CMP     CX,1
         JNZ     LEFT_BAD
RIGH_TBAD:
         CALL    GTBLKX
         JMP     GT_DONE
LEFT_BAD:
         CMP     CX,2
         JNZ     X_ERROR
         CALL    GTXBLKX
         JMP     GT_DONE
```

```
X_ERROR:
          MOV     AX,1
          JMP     GT_DONE
Y_ERROR:
          MOV     AX,1
          JMP     GT_DONE
ALL_OK:   CALL    GTBLK
GT_DONE:
          POP     AX
          POP     BX
          POP     CX
          POP     DX
          POP     DS
          POP     ES
          POP     SI
          POP     DI
          MOV     SP,BP
          POP     BP
          RET
_GET      ENDP

FINDPARAM         PROC      NEAR
          MOV     AX,INXO              ; STORE "XLEFT" (PELS) IN AX AND BX
          MOV     BX,AX
          MOV     CL,RT1
                                       ; BITS 0,1 SPECIFY ONE PEL OUT OF 4
                                       ; GET RID OF THEM TO GET THE BYTE
          SHR     AX,CL                ; COORDINATES
                                       ; STORE THE RESULT IN X0. X0 = THE
          MOV     X0,AX                ; BYTE IN WHICH "XLEFT" (PELS) LIES
          MOV     DX,INX2              ; DX = "XRIGHT" (PELS).
          SUB     DX,BX                ;
          INC     DX
                                       ; SI = DX = XRIGHT - XLEFT + 1 =
          MOV     SI,DX                ;      XLENGTH (PELS).
          MOV     PXLENGTH,DX
          SHR     DX,CL                ; DX = XLENGTH IN BYTES.
          AND     BX,BIT1
          JNZ     LEFT_X
LEFT_OK:                               ; XLEFT IS AT THE BYTE BOUNDARY, SEE
          AND     SI,BIT1              ; IF XRIGHT IS AT THE BYTE BOUNDARY
                                       ; TOO
                                       ; IF XRIGHT IS NOT AT THE BYTE
                                       ; BOUNDARY
          JNZ     RIGHT_X              ; GO TO RIGHT_X.
          SUB     CX,CX                ; THE BLOCK IS AT THE BYTE BOUNDARY
                                       ; SO CX = 0.
                                       ; DX IS EXACT.
          RET
RIGHT_X:                               ; SI = NO. OF PELS IN THE LAST BYTE.
```

```
                                ; SI = 1,2,3 (=0 IS A PREVIOUS CASE).
        MOV     CL,LT1
        SHL     SI,CL
        MOV     AL,[SI+MASK2]
        MOV     LAST_MASK,AL
                                ; DX = NO. OF BYTES NEEDED IN
        INC     DX              ; THE ARRAY.
        MOV     CX,1
        RET
LEFT_X:
        MOV     AX,BIT1
        INC     AL
        SUB     AL,BL
        MOV     CL,LT1
        SHL     AL,CL
                                ; WE USE SHFT_RGT TO SHIFT THE WORD
                                ; IN WHICH A PEL (OTHER THAN ZERO)
                                ; IS THE START OF EACH LINE.
        MOV     BYTE PTR SHFT_RGT,AL
                                ; THE SCREEN BYTE THAT WE WANT TO
                                ; TRANSFER TO THE BUFFER .
                                ; WE SHIFT THE WORD TILL THE DESIRED
                                ; BYTE FITS INTO AL.
                                ; FOR THE LAST BYTE WE NEED ONLY PART
                                ; OF THE BYTE SO WE ZERO THE EXTRA
                                ; PELS USING LAST_MASK.
        AND     SI,BIT1         ; IF SI = 0 WE HAVE NO EXTRA PELS,
        JZ      SET_LST_MSK     ; HENCE THE LAST BYTE IS COUNTED IN
                                ; DX.
        INC     DX              ; INCREMENT DX TO TAKE THE LAST BYTE
                                ; OUT OF DX.
                                ; SINCE SI WAS CALCULATED FROM
                                ; XLENGTH IT INCORPORATED THE EFFECT
                                ; OF X0 AND X2 IN THE LAST BYTE OF
                                ; THE BUFFER. LAST_MASK TAKES CARE OF
                                ; THE EXTRA PELS IN THE LAST BYTE.
                                ; THE DIFFERENCE BETWEEN "GET" AND
                                ; "PUT" FUNCTIONS IS THAT IN "PUT" WE
                                ; WANT TO PRESERVE THE OLD CONTENT OF
                                ; THE SCREEN (i.e. THE EXTRA PELS IN
                                ; THE LAST BYTE). BUT IN "GET" WE
                                ; ZERO THE EXTRA PELS BY LAST_MASK.
SET_LST_MSK:
        MOV     CL,LT1
        SHL     SI,CL
        MOV     AL,MASK2 [SI]   ; SI = NO. OF PELS IN THE LAST BYTE.
        MOV     LAST_MASK,AL    ; SI = 0,1,2,3
        MOV     CX,2
        RET
FINDPARAM       ENDP
```

```
GTBLK   PROC    NEAR
        MOV     CX,2            ; INITIALIZE THE OUTSIDE COUNTER.
LOOP1:
        PUSH    CX              ; STORE THE OUTSIDE COUNTER.
LOOP2:  MOV     CX,DX           ; INITIALIZE THE BYTES COUNTER.
        MOV     AX,DX           ; STORE IT IN AX ALSO.
        SHR     CX,1            ; CX = CX/2 = NO. OF WORDS.
        REPZ    MOVSW           ; MOVE AS WORDS.
        SHR     AX,1            ; IF THE NO. OF BYTES WAS EVEN
        JNB     NEXT_LINE       ; THEN GO TO DO THE NEXT LINE.
        LODSB                   ; NO. OF BYTES WAS ODD SO WE HAVE
        STOSB                   ; TO MOVE THE LAST BYTE.
NEXT_LINE:
        ADD     SI,SORC_INC     ; INCREMENT SI AND DI BY SORC_INC.
        ADD     DI,DX
        DEC     BX              ; IF THEIR IS MORE LINES START AGAIN.
        JNZ     LOOP2
NEXT_BANK:
                                ; REINITIALIZE BX TO THE NO. OF Y
        MOV     BX,LINE_CNTR    ; LINES IN THE SECOND BANK.
        POP     CX              ; RESTORE THE ROUND COUNTER.
        CMP     BX,0            ; DOES THE SECOND BANK HAVE ANY
                                ; LINES ?
                                ; IF NOT, THEN THE BLOCK HAS ONLY
        JBE     GTBLK_DONE      ; ONE Y LINE AND WE ARE DONE.
                                ; ELSE, THE 2ND BANK HAS LINES SO
                                ; CONTINUE.
        MOV     SI,SORC_INDX2   ; SI POINTS TO THE OFFSET IN THE
                                ; SECOND BANK.
        MOV     DI,DEST_INDX2   ; DI POINTS TO THE 2ND LINE IN THE
                                ; BUFFER
CHANG_ORIGN:
                                ; GHANGE FROM EVEN TO ODD BANK
        XOR     SI,02000H       ; OR VICE VERSA.
        LOOP    LOOP1
GTBLK_DONE:     RET
GTBLK           ENDP

GTBLKX  PROC    NEAR
        MOV     CX,2
LOOP1X:
        PUSH    CX
LOOP2X: MOV     CX,DX           ; DX DID NOT INCLUDE THE LAST BYTE
        DEC     CX              ; SO DO THE LAST BYTE OF ADJUST_LAST.
        MOV     AX,CX           ; AX = CX = DX = XLENGTH.
        SHR     CX,1            ; CX = XLENGTH/2 (- 0/1 BYTE).
        REPZ    MOVSW
        SHR     AX,1            ; IF XLENGTH WAS EVEN THEN MOVING
        JNB     ADJUST_LAST     ; DX BYTES IS DONE, GO TO ADJUST_LAST
```

```
        LODSB                     ; DX WAS ODD SO WE STILL HAVE TO MOVE
        STOSB                     ; ONE MORE BYTE.
ADJUST_LAST:
        LODSB                     ; LOAD THE LAST BYTE FROM THE SCREEN.
        MOV     CL,ES:LAST_MASK ; SET TO 0 THE BITS WE DO NOT WANT.
        AND     AL,CL             ; COPY THE BITS, FILL FROM LEFT TO
                                  ; RIGHT.
                                  ; STORE THE RESULT IN THE LAST BYTE
        STOSB                     ; OF THIS LINE.
NEXT_LINEX:
        ADD     SI,SORC_INC
        ADD     DI,DX
        DEC     BX
        JNZ     LOOP2X
NEXT_BANKX:
        MOV     BX,LINE_CNTR
        POP     CX
        CMP     BX,0
        JBE     GTBLKX_DONE
        MOV     SI,SORC_INDX2
        MOV     DI,DEST_INDX2
CHANG_ORGX:
        XOR     SI,02000H
        LOOP    LOOP1X
GTBLKX_DONE:    RET
GTBLKX          ENDP

GTXBLKX PROC    NEAR              ; BEFORE INCREMENTING DL WE HAVE
                                  ; DX = THE NO. OF BYTES EXCEPT
                                  ; THE LAST BYTE 0 <= ( DX = XLENGTH )
                                  ; <= 79 SO DL = DX = XLENGTH - 1.
                                  ; NOW WE HAVE  DL = XLENGTH + 1,
                                  ; DL = 1 IF ONLY THE LAST BYTE TO BE
                                  ; PROCESSED. ( 1 <= DL <= 80. )
        MOV     CX,2
XLOOP1:
        PUSH    CX
        MOV     CL,SHFT_RGT
XLOOP2: MOV     CH,DL
        DEC     CH                ; IF DX WAS ORIGINALLY 0 (i.e. WE
                                  ; HAVE ONLY ONE BYTE, WHICH IS THE
                                  ; LAST ONE) THEN WE HAVE TO MOVE ONLY
                                  ; THIS LAST BYTE SO GO TO LAST_BYTE.
        JZ      XLAST_BYTE
XLOOP3: LODSW                     ; LOAD A WORD FROM THE SCREEN.
        XCHG    AH,AL
                                  ; SHIFT IT TO THE RIGHT TILL THE
        SHR     AX,CL             ; DESIRED BYTE FITS INTO AL.
        STOSB                     ; STORE THIS BYTE INTO THE BUFFER.
                                  ; SI WAS INCREMENTED BY 2 TO GET
```

```
                                  ; BYTES 3,4 (i.e. WE GOT 1,2).
         DEC     SI               ; SO DECREMENT SI
         DEC     CH               ; IF THE BLOCKS ARE DONE THEN END
         JNZ     XLOOP3           ; THE LOOP, IF NOT LOOP AGAIN.
XLAST_BYTE:
         LODSW
         XCHG    AH,AL
         SHR     AX,CL
         AND     AL,ES:LAST_MASK
         STOSB
         DEC     SI
XNEXT_LINE:
         ADD     SI,SORC_INC
         ADD     DI,DX
         DEC     BX
         JNZ     XLOOP2
XNEXT_BANK:
         MOV     BX,LINE_CNTR
         POP     CX
         CMP     BX,0
         JBE     GTXBLK_DONE
         MOV     SI,SORC_INDX2
         MOV     DI,DEST_INDX2
CHNG_XORG:
         XOR      SI,02000H
         LOOP     XLOOP1
GTXBLK_DONE:     RET
GTXBLKX          ENDP
_TEXT            ENDS
END
/* ----------------------- END geth.asm -------------------------- */
```

13.12. File Puth.asm

```
PUBLIC   MASK1,_PUT,FIND_PARAMS_P
DGROUP   GROUP  _BSS,_DATA
         ASSUME  DS:DGROUP
EXTRN    __CHKSTK:NEAR
_DATA    SEGMENT        word    public  'DATA'
MASK1    DB      ?
         DB      07FH,03FH,01FH,00FH,007H,003H,001H
         EVEN
MASK3    DW      07F80H,0FFBFH,0FF9FH,0FF8FH,0FF87H,0FF83H
         DW      0FF81H,0FF80H,03FC0H,0FFDFH,0FFCFH,0FFC7H
         DW      0FFC3H,0FFC1H,0FFC0H,07FC0H,01FE0H,0FFEFH
         DW      0FFE7H,0FFE3H,0FFE1H,0FFE0H,07FE0H,03FE0H
         DW      00FF0H,0FFF7H,0FFF3H,0FFF1H,0FFF0H,07FF0H
         DW      03FF0H,01FF0H,007F8H,0FFFBH,0FFF9H,0FFF8H
         DW      07FF8H,03FF8H,01FF8H,00FF8H,003FCH,0FFFDH
```

```
         DW     0FFFCH,07FFCH,03FFCH,01FFCH,00FFCH,007FCH
         DW     001FEH,0FFFEH,07FFEH,03FFEH,01FFEH,00FFEH
         DW     007FEH,003FEH
         EVEN
PTMODSET DB     0
_DATA    ENDS

_BSS     SEGMENT        word    public  'BSS'
         EVEN
STRNG_MASK
         DW     ?               ; STRNG_MASK AND LAST_MASK ARE
                                ; INITIALIZED IN "FINDPARAMS"
LAST_MASK                       ; FROM THE VALUES IN MASK3
         DW     ?               ; RESPECTIVELY.
                                ; THEY ARE USED IN THE XBLOCK CASE.
RIGHT_MASK                      ; TO BE INITIALIZED IN
         DB     ?               ; "FINDPARAM", FOR THE CASE OF
                                ; "BLOCKX", FROM VALUES IN "MASK1".
LT2      DB     ?
LT3      DB     ?
BIT1     DW     ?
RT1      DB     ?
ADJST1   DB     ?
LT4      DB     ?
_BSS     ENDS

IX0             EQU     [BP+4]
Y0              EQU     [BP+6]
BUFFER          EQU     [BP+8]
DEST_INC        EQU     [BP-2]
SORC_INDX2      EQU     [BP-4]
DEST_INDX2      EQU     [BP-6]
LINE_CNTR_P     EQU     [BP-8]
SHFT_LFT        EQU     [BP-10]
BANK            EQU     [BP-12]  ; BANKS COUNTER.
X0              EQU     [BP-14]

        ODDORG=02000H
_TEXT    SEGMENT BYTE PUBLIC  'CODE'
         ASSUME CS:_TEXT
_PUT     PROC   NEAR
         PUSH   BP
         MOV    BP,SP
         MOV    AX,14
         CALL   __chkstk
         PUSH   DI
         PUSH   SI
         PUSH   ES
         PUSH   DS
         PUSH   AX
```

```
         PUSH    BX
         PUSH    CX
         PUSH    DX
GETSC_MODE:
         MOV     AH,15
         INT     16
         CMP     AL,4
         JNE     HIGH_RES_P
         MOV     RT1,2
         MOV     LT2,1
         MOV     LT3,5
         MOV     BIT1,3
         MOV     ADJST1,16
         MOV     LT4,2
         JMP     FIND_PARAMS_P
HIGH_RES_P:
         CMP     AL,6
         JNE     NOT_GRAPH_P
         MOV     RT1,3
         MOV     LT2,0
         MOV     LT3,4
         MOV     BIT1,7
         MOV     ADJST1,0
         MOV     LT4,1
         JMP     FIND_PARAMS_P
NOT_GRAPH_P:
         JMP     PUT_DONE
FIND_PARAMS_P:
         CALL    FINDPARAM_P
MOVE_SETUP_P:
         MOV     AX,80            ; DEST_INC = 80-SIZE ( = SCREEN_INC.)
         SUB     AX,DX
         MOV     DEST_INC,AX
         MOV     AX,Y0            ;
         SHR     AX,1
         MOV     BL,80
         MOV     DI,DX
         MUL     BL
         MOV     DX,DI
         ADD     AX,X0
         MOV     BX,Y0
         AND     BX,1
         JZ      ORG_EVEN_P
         ADD     AX,ODDORG
         MOV     BX,80
ORG_EVEN_P:      MOV     DI,AX
         ADD     AX,BX            ; BX = 0  OR  80.
         MOV     DEST_INDX2,AX
         MOV     AX,SI            ;
         ADD     AX,DX
```

```
          MOV     SORC_INDX2,AX
          MOV     AX,LINE_CNTR_P
          MOV     BX,AX
          SHR     AX,1              ; AX = NO. OF Y LINES IN THE SECOND
          MOV     LINE_CNTR_P,AX    ; BANK. STORE IT IN THE LINE COUNTER.
          AND     BX,1              ; IF "YLENGTH" WAS ODD THEN BX = 1,
          ADD     BX,AX             ; HENCE THE FIRST BANK HAS ONE MORE
                                    ; LINE THAN THE SECOND.
          CMP     BX,0
          JBE     Y_ERROR_P
          MOV     BYTE PTR BANK,2 ; INITIALIZE THE BANKS COUNTER.
          MOV     AX,0B800H
          MOV     ES,AX
CHOOSE_P:
          JCXZ     G_ALL_OK_P
          CMP     CX,1
          JNZ     G_LEFT_BAD_P
G_RIGHT_BAD_P:
          CALL    PUTBLKX
          JMP     PUT_DONE
G_LEFT_BAD_P:
          CMP     CX,2
          JNZ     X_ERROR_P
          CALL    PUTXBLKX
          JMP     PUT_DONE
X_ERROR_P:
          JMP     PUT_DONE
Y_ERROR_P:
          JMP     PUT_DONE
G_ALL_OK_P:
          CALL    PUTBLK
PUT_DONE:
          POP     DX
          POP     CX
          POP     BX
          POP     AX
          POP     DS
          POP     ES
          POP     SI
          POP     DI
          MOV     SP,BP
          POP     BP
          RET
_PUT      ENDP

FINDPARAM_P       PROC    NEAR
          MOV     SI,BUFFER         ; SI POINTS TO THE BUFFER.
          LODSW                     ;
          MOV     DI,AX             ; DI HERE IS USED AS A GENERAL
                                    ; REGISTER. LET DI="XLENGTH" (PELS).
```

```
         LODSW
                                    ; LET "LINE_CNTR" HOLD THE TOTAL NO.
         MOV    LINE_CNTR_P,AX      ; OF Y LINES, i.e. "YLENGTH".
         MOV    DX,DI               ; DX = "XLENGTH" (PELS).
         MOV    CL,RT1
                                    ; LAST TWO BITS SPECIFY THE EXTRA
                                    ; PELS. GET RID OF THEM.
                                    ; DX = "XLENGTH" (-1 IF WE HAD
         SHR    DX,CL               ; EXTRA PELS.)
         MOV    AX,IX0
         SHR    AX,CL
         MOV    X0,AX
         MOV    BX,IX0
         AND    BX,BIT1             ; BX PELS OF X0 (i.e. BX = 0,1,2,3.)
                                    ; IF THE BLOCK DID NOT START AT PEL 0
         JNZ    LEFT_BAD_P          ; (i.e. WITHIN BYTE) GO TO LEFT_BAD_P
                                    ; DI = NO. OF EXTRA PELS
         AND    DI,BIT1             ; (0,1,2 OR 3 PELS.)
         JNZ    RIGHTX_P            ; EXTRA PELS ? IF SO, GO TO RIGHTX_P.
         SUB    CX,CX               ; NO EXTRA PELS ( DI=0. )
         RET
RIGHTX_P:
         MOV    CL,LT2
         SHL    DI,CL
         MOV    AL,MASK1 [DI]
                                    ; NOTE THAT RIGHT_MASK IS A BYTE, BUT
                                    ; LAST_MASK IS A WORD ( THIS
                                    ; LAST_MASK DIFFERS FROM THE ONE USED
         MOV    RIGHT_MASK,AL       ; IN "GET" FUNCTION. )
         INC    DX
         MOV    CX,1
         RET
LEFT_BAD_P:
                                    ; SHFT_LEFT SHIFTS A BYTE FROM THE
                                    ; BUFFER IN AX TILL IT STARTS AT THE
                                    ; PEL WERE X0 ( OR THE BLOCK )
                                    ; STARTS. STRNG_MASK WILL ZERO THE
                                    ; BITS OF A COPY OF THE SCREEN
                                    ; CORRESPONDING TO THIS BYTE.
                                    ; "ORING" AX AND THE MASKED WORD
                                    ; GIVES US THE CORRECT WORD TO
                                    ; PUT ON THE SCREEN.
         MOV    AX,BIT1
         INC    AL
         SUB    AL,BL
         MOV    CL,LT2
         SHL    AL,CL
         MOV    SHFT_LFT,AL
         AND    DI,BIT1
         JZ     SET_LST_MSK_P
```

```
         INC     DX
SET_LST_MSK_P:
         DEC     BX
         MOV     CL,LT3
         SHL     BX,CL
                                 ; BX <= 7*16 =112 SO WE CAN USE BL
         ADD     BL,ADJST1       ; INSTEAD OF BX.
         MOV     CL,LT4
         SHL     DI,CL
         MOV     AX,MASK3[BX+DI] ; THE LAST_MASK HAS ZERO BITS
                                 ; STARTING AT THE PEL OF X0
                                 ; ( DEFINED BY BX. ) AND CONTINUES
                                 ; FOR THE NO. OF EXTRA BITS
         MOV     LAST_MASK,AX    ; ( DEFINED BY DI. )
         MOV     AX,MASK3 [BX]
         MOV     STRNG_MASK,AX
         MOV     CX,2
END_FIND_P:      RET
FINDPARAM_P      ENDP

PUTBLK           PROC    NEAR
LOOP1:
LOOP2_P: MOV     CX,DX
         MOV     AX,DX
         SHR     CX,1
         REPZ    MOVSW
         SHR     AX,1
         JNB     NEXT_LINE_P
         LODSB
         STOSB
NEXT_LINE_P:
         ADD     DI,DEST_INC
         ADD     SI,DX
         DEC     BX
         JNZ     LOOP2_P
NEXT_BANK_P:
         MOV     BX,LINE_CNTR_P
         CMP     BX,0
         JZ      PUTBLK_DONE
         MOV     DI,DEST_INDX2
         MOV     SI,SORC_INDX2
CHNG_ORG_P:
         XOR     DI,02000H
         DEC     BYTE PTR BANK
         JNZ     LOOP1
PUTBLK_DONE:     RET
PUTBLK           ENDP

PUTBLKX          PROC    NEAR
LOOP1X_P:
```

```
LOOP2X_P:
         MOV     CX,DX
         DEC     CX
         JZ      ADJST_LASTX_P
         MOV     AX,CX
         SHR     CX,1
         REPZ    MOVSW
         SHR     AX,1
         JNB     ADJST_LASTX_P
         LODSB
         STOSB
ADJST_LASTX_P:                   ; WE DO NOT CHECK IF LAST BYTE IS
                                 ; FULL BECAUSE THAT CASE IS HANDLED
                                 ; IN "BLOCK" AND NOT "BLOCKX".
         LODSB
         MOV     AH,ES:[DI]
         AND     AH,RIGHT_MASK
         OR      AL,AH
         STOSB
NEXT_LINEX_P:
         ADD     DI,DEST_INC
         ADD     SI,DX
         DEC     BX
         JNZ     LOOP2X_P
NEXT_BANKX_P:
         MOV     BX,LINE_CNTR_P
         CMP     BX,0
         JZ      PUTBLKX_DONE
         MOV     DI,DEST_INDX2
         MOV     SI,SORC_INDX2
CHNG_ORGX_P:
         XOR     DI,02000H
         DEC     BYTE PTR BANK
         JNZ     LOOP1X_P
PUTBLKX_DONE:    RET
PUTBLKX          ENDP
PUTXBLKX         PROC    NEAR
         MOV     CL,SHFT_LFT
XLOOP1_P:
         MOV      CH,DL
         PUSH     DX
         DEC      CH             ; WE DID NOT NEED THIS IN "BLOCKX"
                                 ; BECAUSE REP_STRING WILL TAKE CARE
                                 ; OF IT AS FOLLOWS : DX=0 SO "REPZ"
                                 ; WILL NOT MOVE ANYTHING. SINCE ZERO
                                 ; IS AN EVEN NUMBER THE PROGRAM WILL
                                 ; JUMP TO THE NEXT LINE WITHOUT
                                 ; MOVING AN EXTRA BYTE.
         JZ      XLAST_BYTE_P
XLOOP2_P:
```

```
         XOR     AH,AH
         LODSB
         SHL     AX,CL
         XCHG    AH,AL
         MOV     DX,ES:[DI]
         AND     DX,STRNG_MASK
         OR      AX,DX
         STOSW
         DEC     DI
         DEC     CH
         JNZ     XLOOP2_P
XLAST_BYTE_P:
         XOR     AH,AH              ; FILL AH WITH ZEROS.
         LODSB                      ; AL = BYTE FROM THE BUFFER THAT NEED
                                    ; TO BE PUT ON THE SCREEN STARTING AT
                                    ; THE PEL X0.
         SHL     AX,CL              ; THE SHIFT WILL PUT IT IN AX AT
         XCHG    AH,AL              ; THE SAME PLACE. THE OTHER BITS IN
                                    ; AX WILL BE ZEROS.
         MOV     DX,ES:[DI]
         AND     DX,LAST_MASK       ; DX = ZEROS IN THAT PART OF THE
                                    ; BYTE, OTHER BITS ARE SET TO ONES.
         OR      AX,DX              ; AX = NEW SCREEN WORD. PUT IN
                                    ; PLACE WITHOUT CHANGING OTHER BITS.
         STOSW                      ; PUT THE WORD ON THE SCREEN.
         DEC     DI                 ; ADJUST DI TO PUT THE NEXT BYTE.
                                    ; ( SAY DO WORD 1+1/2 INSTEAD OF
                                    ; WORD 2. )
NEXT_XLINE_P:
         ADD     DI,DEST_INC
         POP     DX
         ADD     SI,DX
         DEC     BX
         JNZ     XLOOP1_P
XNEXT_BANK_P:
         MOV     BX,LINE_CNTR_P
         CMP     BX,0
         JZ      PUTXBLK_DONE
         MOV     DI,DEST_INDX2
         MOV     SI,SORC_INDX2
CHNG_XORG_P:
         XOR     DI,02000H
         DEC     BYTE PTR BANK
         JNZ     XLOOP1_P
PUTXBLK_DONE:    RET
PUTXBLKX         ENDP
_TEXT            ENDS
                 END
/* ------------------------- END puth.asm ------------------------ */
```

13.13. File Swap.asm

```
; SWAP LOW AND HIGH BYTES IN EACH WORD
NAME            SWAP
TITLE           SWAP BYTES IN EACH WORD
DGROUP  GROUP   CONST, _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
PUBLIC  _swapbyts
FROMADDRS       EQU     [BP+4]  ; PARAMETERS PASSED.
TOADDRS         EQU     [BP+6]
WORDCONT        EQU     [BP+8]
_TEXT           SEGMENT

_swapbyts       PROC    NEAR
         PUSH   BP              ; SAVE REGISTERS.
         MOV    BP,SP
         PUSH   DI
         PUSH   SI
                                ; PUT THE NUMBER OF WORDS TO BE
         MOV    CX,WORDCONT     ; SWAPPED IN CX.
         MOV    SI,FROMADDRS    ; SOURCE OPERAND IS ADDRESSED BY SI.
                                ; DESTINATION OPERAND IS
         MOV    DI,TOADDRS      ; ADDRESSED BY DI.
                                ; LOOP UNTIL THE NUMBER OF WORDS
LOOP1:                          ; IN CX BECOMES ZERO.
                                ; TRANSFER A WORD FROM THE
                                ; SOURCE  [( SI)] TO AX, THEN
         LODSW                  ; LET SI=SI+2.
         XCHG   AH,AL           ; SWAP THE LOW AND HIGH BYTES

                                ; TRANSFER A WORD OPERAND FROM
                                ; AX TO DESTINATION ( DI )
         STOSW                  ; THEN LET DI=DI+2.

                                ; FIRST LET CX=CX-1 THEN
         LOOP   LOOP1           ; IF CX=0, EXIT LOOP1.
         POP    SI              ; RESTORE THE REGISTERS.
         POP    DI
         MOV    SP,BP
         POP    BP
         RET
_swapbyts       ENDP
_TEXT   ENDS
END
/* ---------------------- END swap.asm --------------------------- */
```

13.14 File Mtchbts.asm

```
NAME    MTCHBITS
TITLE   TO MATCH PASSED BITS TO A PATTERN IN APPROPRIATE ARRAYS.
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
_DATA   SEGMENT  WORD PUBLIC 'DATA'
EXTRN   _LEFTBITSWORD:WORD
_DATA   ENDS
PUBLIC          _match_all_bits
WORD            EQU     [BP+4]  ; PASSED PARAMETERS.
COLORARRAY      EQU     [BP+6]
CODEARRAY       EQU     [BP+8]
GROUPARRAY      EQU     [BP+10]
CLRBITSPTR      EQU     [BP+12]
CODEBITSPTR     EQU     [BP+14]
GROUPCOUNT      EQU     [BP-2]

_match_all_bits      PROC         NEAR
         PUSH   BP
         MOV    BP,SP
         PUSH   DI
         PUSH   SI
         PUSH   ES
         PUSH   DS
         POP    ES
         MOV    DI,COLORARRAY
         MOV    BX,GROUPARRAY
         MOV    DX,[BX]             ; PUT THE NUMBER OF GROUPS IN THE
                                    ; COUNTER DX.
LOOP1:
                                    ; ADVANCE INDEX (BX) TO THE FIRST
         ADD    BX,2                ; ELEMENT OF PAIRS IN "GROUPARRAY".
                                    ; GET THE LENGTH IN BITS OF THE
         MOV    SI,[BX]             ; CODES TO LOOK FOR.
                                    ; MULTIPLY BY 2 TO GET THE INDEX
                                    ; OF THE MASK IN BYTES SINCE
                                    ; MASK IS AN ARRAY OF UNSIGNED
         SHL    SI,1                ; NUMBERS ( i.e. WORDS )
         MOV    AX,WORD             ; COPY THE WORD WE ARE LOOKING FOR.
                                    ; "LEFTBITSWORD" IS THE MASK. IF
                                    ; SI IS EQUAL TO 3 FOR EXAMPLE
                                    ; THEN ONLY THE 3 MOST LEFT
                                    ; BITS ARE NOT MASKED WHILE THE
                                    ; REMAINING BITS ARE SET TO ZEROS.
         AND    AX,_LEFTBITSWORD[SI]
         ADD    BX,2                ; ADVANCE "GROUPARRAY" INDEX TO
                                    ; GET THE SECOND ELEMENT OF THE
                                    ; CURRENT PAIR WHICH TELLS THE NUMBER
                                    ; OF CODES IN "COLORARRAY" THAT
```

```
         MOV    CX,[BX]         ; HAS THE SAME NUMBER OF BITS.
                                ; KEEP SCANNING FOR A MATCH
                                ; WITH THE WORD IN AX UNTIL MATCHED
         REPNE  SCASW           ; OR CX IS DECREMENTED TO ZERO.
                                ; z=0 MEANS THAT CX WAS
                                ; DECREMENTED TO ZERO AND THUS
         JNE    NO_MATCH        ; NO MATCH OCCURRED.
                                ; BY REACHING THIS POINT THE
                                ; z FLAG WAS NOT SET TO ZERO
                                ; AND THUS A MATCH OCCURRED.
         MOV    AX,[BX-2]       ; SO THE LENGTH OF THE CODE IS
         MOV    SI,CODEBITSPTR  ; RETURNED IN THE WORD POINTED
         MOV    [SI],AX         ; TO BY "CODEBITSPTR".
                                ; TO GET THE INDEX OF THE MATCHED
                                ; PATTERN IN "COLORARRAY" SUBTRACT
                                ; THE CURRENT POSITION FROM THE BASE
         SUB    DI,COLORARRAY   ; OR THE HEAD OF THE ARRAY.
         MOV    BX,CODEARRAY    ; FIND THE CODE IN "CODEARRAY" OF
         MOV    AX,[BX+DI-2]    ; THE SAME INDEX IN "COLORARRAY".
         MOV    SI,CLRBITSPTR   ; THE RUN LENGTH IS RETURNED IN THE
         MOV    [SI],AX         ; WORD POINTED TO BY "CLRBITSPTR".
         MOV    AX,1            ; THE RETURNED VALUE OF FUNCTION = 1.
         JMP    DONE
NO_MATCH:
         DEC    DX              ; DECREMENT THE GROUPS COUNTER
                                ; IF THERE ARE MORE GROUPS
         JNZ    LOOP1           ; GO TO LOOP1 TO PROCESS THEM.
         SUB    AX,AX           ; RETURNED VALUE OF FUNCTION = 0.
DONE:    POP    ES
         POP    SI
         POP    DI
         MOV    SP,BP
         POP    BP
         RET
_match_all_bits ENDP
_TEXT           ENDS
END
/* ---------------------- END mtchbts.asm ------------------------ */
```

14. APPENDIX C. PROGRAM LISTINGS OF THE CODE OF THE CCITT TWO DIMENSIONAL COMPRESSION TECHNIQUE

The files in this listing make use of the files in the following sections:

- Appendix B: 13.4. and 13.7. - 13.14.

14.1. File Main.c

```
/*
 * The heading and comments are the same
 * as those in file main.c appendix B section 13.1.
 */

#include        <stdio.h>
#include        <memory.h>
#include        <dos.h>
#include        <io.h>
#include        <fcntl.h>
#include        <malloc.h>
#define         LINT_ARGS
#define         SCREENSIZE      16384
#define         XMAX            640
#define         YMAX            200
#define         HI_RES          6
#define         TEXT_MODE       3
#define         ulong           unsigned  long

void            get(int,int,int,int,char *);
unsigned        cmprs_line(char *);
float           get_avgfactor();
void            print_results( char *, unsigned, unsigned, unsigned,
                               unsigned, unsigned, unsigned, float);

static          int             x1,y1,x2,y2;
                char            datafile[41];
static          char            scrfilebufr[4+(XMAX/8)*(YMAX)];
static          unsigned        cmprsbufr[XMAX][(YMAX+32)/16];
static          char            *uncmprsbufr;

main(argc,argv)
int     argc;
char    *argv[];
{
unsigned                        xsizeinbytes,xsize;
register        unsigned        i,ysize;

if( argc < 6 )
   {
        printf("enter x1 y1 x2 y2 \n");
        scanf("%d %d %d %d",&x1,&y1,&x2,&y2);
        while((getchar())!='\n')
```

```
                    ;
   }
else
   {
        x1=atoi(argv[2]); y1=atoi(argv[3]);
        x2=atoi(argv[4]); y2=atoi(argv[5]);
   }
if( argc > 1 )
        strcpy( datafile, argv[1] );
                                   /* Read the data from the input */
init_screen(argc);                 /* file and dump it to the      */
                                   /* screen.                      */
uncmprsbufr= scrfilebufr;
uncmprsbufr+=4;                    /* Skip over "xsize" and "ysize"*/
get(x1,y1,x2,y2,(char *)scrfilebufr);
for(i=0;i<=2000;i++)  ;            /*  A delay loop.               */
setscmode(TEXT_MODE);
ysize=y2-y1+1;
xsize=x2-x1+1;
xsizeinbytes= (xsize/8)+((xsize%8)>0) ;
                                   /* First  two  numbers  in  the */
*(unsigned *)scrfilebufr=xsize;    /* "screenfilebufr" represent   */
                                   /* the width and the height of  */
                                   /* the block.                   */
*(unsigned *)(scrfilebufr+2)=ysize;
printf("starting to compress ");
init_cmprsdblk((unsigned *)cmprsbufr);
init_uncmprsdblk((scrfilebufr+4),xsize,ysize);
init_line_parm(xsize);
cmprs_blk_2d();
memset((scrfilebufr+4),'\0',16000);
printf(" starting to decompress \n");
init_dcmprsbfr((scrfilebufr+4),xsize);
init_cmprs(cmprsbufr);
init_dcmprs_blk_2d(xsize,ysize,scrfilebufr+4);
dcmprs_blk_2d();
                                   /* If no  argument was  entered */
if( argc < 2 )                     /* at  the  command  line then  */
   {                               /* display the data to the      */
                                   /* screen.                      */
        setscmode(HI_RES);
        put(x1,y1,scrfilebufr);
        getchar();
        setscmode(TEXT_MODE);
   }
print_results(  datafile, x1, y1, x2, y2, get_cmprstime(),
                get_dcmprstime(), get_avgfactor() );
}
/*------------------------ END  MAIN  ----------------------------*/
/*----------------------- END main.c ----------------------------*/
```

14.2. File Cmprs2d.c

```
/*
 * Refer to file "cmprsln.c" in appendix B section 13.2
 * for comments on functions and variables.
 */

#include        <stdio.h>
#include        <v2tov3.h>
#include        <malloc.h>
#include        <memory.h>
                                   /* "KFACTOR"-1 = maximum number */
                                   /* of lines coded in  the 2-d   */
                                   /* code after coding in the 1-d.*/
#define         KFACTOR          2
#define         BLACKCHAR       '0'
#define         WHITECHAR       '1'
#define         BLACK            0
#define         WHITE            1
#define         switch_a0_a1_colors {tmpcolorchar=a0colorchar;\
                                        a0colorchar=a1colorchar;\
                                        a1colorchar=tmpcolorchar ;}
unsigned        gttime();
float           get_avgfactor();
unsigned        get_cmprs_reslt();
unsigned        cmprs_line_1d();
void            set_cmprscontr_to_zero();
void            init_uncmprsdblk(char *, unsigned, unsigned);
void            update_cmprsdblk(unsigned,int);
void            updt_cmprsblk_code(unsigned,int);

static          unsigned        *uncmprsdwordptr;
static          unsigned        nmbrlines,xsize,xmaxpls1;
static          unsigned        evenxsize,xsizeinbytes;
static          unsigned        cmprstime;
static          char            *prvslinestart;
static          unsigned long   totalcmprsbits = 0;

/* ======================= CMPRS_BLK_2D ========================== */
/* This function compresses a block of the screen using MREAD      */
/* standard. For complete description of the details of MREAD see */
/* section 4.3. Each line is assumed to be from pel 1 to pel        */
/* "xsize". Pel 0 is an imaginary pel before the line. Pel          */
/* "xmaxpls1" is an imaginary pel after the line.                   */
/* Black changing element means first black pel after a run of     */
/* white pels.                                                      */
/* a0 : The reference or starting changing element in the coding   */
/*      line.  At the start of the coding line, a0 is initialized  */
```

```
/*      to an imaginary black changing element at pel 0.          */
/* a1 : The next changing element to the right of a0 on the coding */
/*      line. This has an opposite color of a0.                   */
/* a2 : The next changing element to the right of a1 on the coding */
/*      line.                                                     */
/* b1 : The next  changing  element on the  reference  line to the */
/*      right of a0 and having the same color as a1.              */
/* b2 : The next  changing  element on the  reference line to the  */
/*      right of b1.                                              */
/* If any of the coding elements a1, a2, b1, b2 is  not detected   */
/* at any time during the coding of the line,  then it  is set  to */
/* pel "xmaxplsl".                                                  */
/* ============================================================== */
void    cmprs_blk_2d()

{
unsigned        i,j;                /* Loop  counters.             */
char            *refrenceline;
char            *codeline;
char            *tmpptr;
register        unsigned        a0,a1;
unsigned        b1,b2,k;
unsigned        a2,a0a1,a1a2;
unsigned        tstart,tend;
int             a0color,a1color,tmpcolor;
char            a0colorchar,a1colorchar,tmpcolorchar;

tstart=gttime();
refrenceline=malloc(xsize+2);
codeline=malloc(xsize+2);
                                    /* k  should be  set to  zero   */
                                    /* before we  enter the  loop   */
                                    /* and  thus the  first  line   */
                                    /* ( reference  line )  would   */
                                    /* be One-Dimensionally coded.  */
k=0;
                                    /* This initialization is needed*/
                                    /* so that the first search for */
refrenceline[0]=BLACKCHAR;          /* b1 works correctly.          */
                                    /* Keep  looping  until all the */
for(i=1; i <= nmbrlines; i++)       /* lines in the page or the     */
     {                              /* screen  are processed.       */
     if(k != 0)                     /* Is this line to be 2-d coded?*/
         {                          /* Line should be 2-d coded.    */
         set_cmprscontr_to_zero();
         swapbits_to_string(uncmprsdwordptr,codeline+1,xsize);
         swapbits_to_string(prvslinestart,refrenceline+1,xsize);
         a0 = 0;
         a0colorchar=BLACKCHAR;
                                    /* Loop while not end of line.  */
```

```
while( a0 < xmaxpls1 )
     {                    /*    Detect "a1color".         */
     a1colorchar = ( a0colorchar == WHITECHAR ?
                              BLACKCHAR : WHITECHAR);
                          /*   Detect a1.                 */
                          /* To detect a1, a2, b1, and b2 */
                          /* we equate the number of bytes*/
                          /* we search to ( "xmaxpls1" -  */
                          /* index  of the 1st byte to be */
                          /* searched.) This is equivalent*/
                          /* to [(xsize-index of 1st byte */
                          /* to be searched ) + 1 ].      */
     if(tmpptr=memchr(&codeline[a0+1],a1colorchar,
                                       xmaxpls1-a0))
          a1=tmpptr-codeline;
     else
          a1=xmaxpls1;
     while(1)
          {
                          /*      Detect  b1.             */
          if( refrenceline[a0] == a1colorchar )
               {
                          /* Pel  refrenceline[a0]  has   */
                          /* the same color as a1 then pel*/
                          /* refrenceline[a0+1]  can't    */
                          /* be a  changing  element  of  */
                          /* "a1color". Hence :           */
                          /* (1) search for the first     */
                          /* changing element of "a0color"*/
               if(tmpptr=memchr(&refrenceline[a0+1],
                               a0colorchar,xmaxpls1-a0))
                    {
                          /* (2) search for the first     */
                          /* changing element of "a1color"*/
                          /* after "tmpptr".              */
                    b1=tmpptr-refrenceline;
                    if(tmpptr=memchr(tmpptr+1,a1colorchar,
                                       xmaxpls1-b1))
                         b1=tmpptr-refrenceline;
                    else
                         b1=xmaxpls1;
                    }
               else
                    b1=xmaxpls1;
          }
     else
          {
                          /* Pel refrenceline[a0] has the */
                          /* same  color as a0, then  pel */
                          /* refrenceline[a0+1] can  be a */
```

```
                /* changing element of "a1color"*/
                /* Hence find it.               */
if(tmpptr=memchr(&refrenceline[a0+1],
                     a1colorchar,xmaxpls1-a0))
     b1=tmpptr-refrenceline;
else
     b1=xmaxpls1;
}
                /*        Detect  b2.           */
if(tmpptr=memchr(&refrenceline[b1+1],
                     a0colorchar,xmaxpls1-b1))
     b2=tmpptr-refrenceline;
else
     b2=xmaxpls1;
                /* If b2 < a1 then we have to   */
                /* do pass mode coding.  Thus   */
                /* this mode is identified when */
                /* the position of b2 lies to   */
                /* the left of a1. The purpose  */
                /* of this mode is to identify  */
                /* the white or black runs on   */
                /* the reference line which are */
                /* not adjacent to the corres-  */
                /* ponding white or black runs  */
                /* on the coding line.          */
if( b2 < a1 )
     { updt_cmprsblk_code(0x1,4); a0=b2;}
else
     {
     if(abs((int)a1-(int)b1)<=3)
          {     /* Vertical mode coding : when  */
                /* this mode is identified, the */
                /* position of a1 is coded      */
                /* relative to the the position */
                /* of b1. The relative distance */
                /* a1b1 can take one of seven   */
                /* values each of which is      */
                /* represented by a separate    */
                /* codeword.                    */
          switch((int)(a1-b1))
               {
                     /* a1 to the left of b2 by */
                     /* 3 bits.                 */
               case -3:{
                        updt_cmprsblk_code(0x2,7);
                        break;
                       }
                     /* a1 to the left of b2 by */
                     /* 2 bits.                 */
               case -2:{
```

```
                   updt_cmprsblk_code(0x2,6);
                   break;
                  }
                /* a1  to  the  left  of   */
                /* b2  by  1  bits.        */
          case -1:{
                   updt_cmprsblk_code(0x2,3);
                   break;
                  }
                /* a1  just  under  b1.    */
          case  0:{
                   updt_cmprsblk_code(0x1,1);
                   break;
                  }
                /* a1  to  the  right  of  */
                /* b2  by  1  bit.         */
          case  1:{
                   updt_cmprsblk_code(0x3,3);
                   break;
                  }
                /* a1  to  the  right  of  */
                /* b2  by  2  bits.        */
          case  2:{
                   updt_cmprsblk_code(0x3,6);
                   break;
                  }
                /* a1  to  the  right  of  */
                /* b2  by  3  bits.        */
          case  3:{
                   updt_cmprsblk_code(0x3,7);
                   break;
                  }
          default:printf(
                    "error in vertical \n");
          }
     a0=a1;
     switch_a0_a1_colors  /* --- MACRO --- */
     }
else
     {     /* Horizontal  Mode  Coding :   */
           /* If the vertical mode coding  */
           /* can't  be  used to  code the */
           /* position  of  a1,  then its  */
           /* position  must be  coded  by */
           /* the horizontal mode coding.  */
           /*    Detect    a2.             */
     if(tmpptr=memchr(&codeline[a1+1],
                aOcolorchar,xmaxpls1-a1))
          a2=tmpptr-codeline;
     else
```

```
                                   a2=xmaxpls1;
                             a0a1=a1-a0;
                                    /* If the horizontal mode coding*/
                                    /* is  used to  code the  first */
                                    /* element on the coding line,  */
                                    /* then  the  value  of a0a1 is */
                                    /* replaced by a0a1-1 to ensure */
                                    /* that the correct run-length  */
                                    /* value is transmitted, because*/
                                    /* the first element was not    */
                                    /* real but an imaginary black  */
                                    /* changing  element.           */
                             if( a0 == 0 )
                                   a0a1 -=1;
                                    /* Flag  "codeword"  of the     */
                                    /* horizontal  mode = '0001'.   */
                             updt_cmprsblk_code(0x1,3);
                             update_cmprsdblk(a0a1,
                                               a0colorchar-BLACKCHAR);
                             update_cmprsdblk(a2-a1,
                                               a1colorchar-BLACKCHAR);
                             a0=a2;
                         }
                         break;
                    }
               }
          }
          k--;
          totalcmprsbits+=get_cmprs_reslt();
          uncmprsdwordptr = (unsigned *)
                        ((char  *)uncmprsdwordptr + xsizeinbytes);
    }
    else
    {                                /* k = 0, so the  current  line */
                                     /* should  be  coded  by  the   */
                                     /* One-Dimensional  coding      */
                                     /* algorithm.                   */
          totalcmprsbits+=cmprs_line_1d();
          k = KFACTOR-1;
    }
                                     /* Update "prvslinestart" to    */
                                     /* point to the start of the    */
                                     /* next line.                   */
prvslinestart += xsizeinbytes ;
}
free(refrenceline);
free(codeline);
tend=gttime();
if(tend>tstart)
        cmprstime=tend-tstart;
```

```
else
        cmprstime=(6000-tstart)+tend;
}
/* ----------------------- END CMPRS_2D() ---------------------- */

/* ====================== init_uncmprsdblk ==================== */
/* initialize local variables.                                  */
/* ============================================================ */
void    init_uncmprsdblk(blockstart,xsizein,ysizein)
char            *blockstart;
unsigned        ysizein,xsizein;

{
xsize=xsizein;
uncmprsdwordptr=(unsigned *) blockstart;
xsizeinbytes=(xsize/8)+((xsize%8)>0);
prvslinestart=blockstart - xsizeinbytes ;
nmbrlines=ysizein;
xmaxpls1=xsize+1;
                                /* The part of the line that */
                                /* corresponds to words given */
evenxsize=( ( (xsize/8) /2) *2);  /* in bytes.                */
}
/* -------------------- END init_uncmprsdblk ------------------- */

#include        <dos.h>
#define         LINT_ARGS
#define         BLACKBIT        0
#define         WHITEBIT        1
#define         ENDBITS         2

unsigned        get_cmprs_reslt();
void            init_lastbits(unsigned);
void            init_cmprsdblk(unsigned *);
void            update_cmprsdblk(unsigned,int);
void            cmprs_lastbits(unsigned,unsigned,int);
void            swapbyts(unsigned *,unsigned *,unsigned);

static unsigned         lastbits,nmbrwords;

/*===================== cmprs_line() ===========================*/
unsigned        cmprs_line (oldlineptr)
char            *oldlineptr;

{
extern unsigned *uncmprsdwordptr;
unsigned        *currentword;
int             wordcount;
int             color,lastcolor,bitcolor;
unsigned        bitcontr=0;
```

```
register        unsigned        word,bitpos;

wordcount=nmbrwords;                /*  Initialize the variables.  */
set_cmprscontr_to_zero();
swapbyts( uncmprsdwordptr, uncmprsdwordptr, nmbrwords);
word=*uncmprsdwordptr;
if ((word)&0x8000)                  /* Is bit 16 in "word"  white ? */
        {                           /* Yes, bit 16 was white.      */
        update_cmprsdblk(0,BLACKBIT);
        color=WHITEBIT;
        }
else
        {                           /* Bit 16 was black.           */
        color=BLACKBIT;
                                    /* Negate the word so we can   */
        word=~word;                 /* check for the new color.    */
        }
                                    /* We assume xsize >= 16, to   */
                                    /* take care of xsize < 16. We */
bitpos=16;                          /* have to modify the code here.*/
while(color<ENDBITS)                /* While not end of line.      */
        {
                                    /* While color is the same and */
                                    /* we are still inside the     */
                                    /* current word.               */
        while( (word&0x8000) && (bitpos > 0) )
                {
                bitcontr++;
                bitpos--;           /* Bit position in a word.     */
                word=word<<1;       /* Get the next bit in bit 16.  */
                }
        if(bitpos > 0)              /* Still inside current word ?  */
                {
                update_cmprsdblk(bitcontr,color);
                word=~word;
                color=(color) ? 0 : 1;
                bitcontr=0;
                }
                                    /* Done  with  all  bits  in   */
        else                        /* current word.               */
                {
                bitpos=16;          /* Start  again  with  bit 16  */
                uncmprsdwordptr++;  /* of  the  next  word.        */

                                    /* If the color is black then  */
                                    /* negate the word pointed to by*/
                                    /* "uncmprsdwordptr" to  check */
                                    /* for the color later.        */
                word=(color) ? *uncmprsdwordptr :
                                            ~(*uncmprsdwordptr);
```

```
                                   /* Test for the end of the line */
                                   /* marker.                      */
                if(--wordcount == 0)
                        {          /* Save the last color in this  */
                                   /* line.                        */
                        lastcolor=color;
                                   /* Signal eol to the outer loop.*/
                        color=ENDBITS;
                        }
                }
        }
if(lastbits == 0)                  /* Does the line fit in the word*/
                                   /* boundary ?                   */
        update_cmprsdblk(bitcontr,lastcolor);
else
        cmprs_lastbits(*uncmprsdwordptr,bitcontr,lastcolor);
if(color>ENDBITS)
        printf(" ****** error in color, color=%d /n",color);
                                   /* Return the number of bits    */
return(get_cmprs_reslt());         /* in that compressed line.     */
}
/* ---------------------- END cmprs_line() ----------------------*/

/*===================== init_line_param() =======================*/
/* Initialize some static variables to the appropriate values.   */
/*===============================================================*/
void init_line_parm(xsize)
unsigned        xsize;

{
nmbrwords=xsize/16;
lastbits=xsize & 0x000f;           /* Let lastbits = xsize % 16.   */
init_lastbits(lastbits);
}
/* ---------------------- END init_line_param() ------------------ */

/* ==================== get_cmprstime() ======================== */
unsigned        get_cmprstime()
{
return (cmprstime) ;
}
/* ------------------- END get_cmprstime() ---------------------- */

/* ==================== get_avgfactor() ======================== */
float           get_avgfactor()
{
return ((float) ( (float) xsize* (float) nmbrlines/totalcmprsbits ));
}
/* ------------------ END get_avgfactor() ----------------------- */
/* --------------------- END cmprs2d.c ------------------------- */
```

14.3. File Cupdt.c

```
/*
 *==========================================================
 * STATIC VARIABLES :
 *
 * bitsleft      : Number of bits still vacant in "cmprsword", it
 *                 starts with 16 bits left in the word.
 * cmprscounter  : Count number of the bits in the compressed block
 *                 which is filled from left to right.
 * cmprsdwordptr: Pointer to the  current  word  position  in  the
 *                 compressed block.
 *==========================================================
 */

static  int             bitsleft;
static  unsigned        cmprscounter;
static  unsigned        *cmprsdwordptr;

/*==================== UPDATE_CMPRSDBLK() ====================*/
/* This is the function update_cmprsdblk( bitcounter, color),    */
/* where "bitcounter" is the number of consecutive bits of the   */
/* current color.                                                */
/*===========================================================*/
void    update_cmprsdblk(uncmprsdbitscont,color)
unsigned        uncmprsdbitscont;
register        int     color;

{
   struct       FAXDATA
                {
                                /* Code for a sequence of bits  */
                                /* of type color and the run-   */
                                /* length = the no. of the      */
                                /* uncompressed bits.           */
                        unsigned   bits;
                                /* Length of the code in the    */
                                /* bits.                        */
                        int        length;
                };

                                /* Initialize FAX. FAX[0][] ==  */
                                /* black data , FAX[1][] ==     */
                                /* white data.                  */
static  struct  FAXDATA FAX[2][74]={ {
        0x35,8, 0x7,6,  0x7,4,  0x8,4,  0xb,4,  0xc,4,  0xe,4,
        0xf,4,  0x13,5, 0x14,5, 0x7,5,  0x8,5,  0x8,6,  0x3,6,
        0x34,6, 0x35,6, 0x2a,6, 0x2b,6, 0x27,7, 0xc,7,  0x8,7,
```

```
        0x17,7, 0x3,7,  0x4,7,  0x28,7, 0x2b,7, 0x13,7, 0x24,7,
        0x18,7, 0x2,8,  0x3,8,  0x1a,8, 0x1b,8, 0x12,8, 0x13,8,
        0x14,8, 0x15,8, 0x16,8, 0x17,8, 0x28,8, 0x29,8, 0x2a,8,
        0x2b,8, 0x2c,8, 0x2d,8, 0x4,8,  0x5,8,  0xa,8,  0xb,8,
        0x52,8, 0x53,8, 0x54,8, 0x55,8, 0x24,8, 0x25,8, 0x58,8,
        0x59,8, 0x5a,8, 0x5b,8, 0x4a,8, 0x4b,8, 0x32,8, 0x33,8,
        0x34,8, 0x1b,5, 0x12,5, 0x17,6, 0x37,7, 0x36,8, 0x37,8,
        0x64,8, 0x65,8, 0x68,8, 0x67,8} , {

        0x37,10, 0x2,3,   0x3,2,   0x2,2,   0x3,3,   0x3,4,
        0x2,4,   0x3,5,   0x5,6,   0x4,6,   0x4,7,   0x5,7,
        0x7,7,   0x4,8,   0x7,8,   0x18,9,  0x17,10, 0x18,10,
        0x8,10,  0x67,11, 0x68,11, 0x6c,11, 0x37,11, 0x28,11,
        0x17,11, 0x18,11, 0xca,12, 0xcb,12, 0xcc,12, 0xcd,12,
        0x68,12, 0x69,12, 0x6a,12, 0x6b,12, 0xd2,12, 0xd3,12,
        0xd4,12, 0xd5,12, 0xd6,12, 0xd7,12, 0x6c,12, 0x6d,12,
        0xda,12, 0xdb,12, 0x54,12, 0x55,12, 0x56,12, 0x57,12,
        0x64,12, 0x65,12, 0x52,12, 0x53,12, 0x24,12, 0x37,12,
        0x38,12, 0x27,12, 0x28,12, 0x58,12, 0x59,12, 0x2b,12,
        0x2c,12, 0x5a,12, 0x66,12, 0x67,12, 0xf,10,  0xc8,12,
        0xc9,12, 0x5b,12, 0x33,12, 0x34,12, 0x35,12, 0x6c,13,
        0x6d,13, 0x4a,13} } ;

register unsigned code;            /* Code of the run of the pels. */
int               length;          /* Length of the above code     */
unsigned          multiple;        /*  = "uncmprsdbitscont" / 64.  */
unsigned          bitcont;         /*  Local run-length.           */

                                   /* To get the least significant */
static  unsigned  mask1=0x003f;    /* 6 bits.                      */

                                   /* Is uncmprsdbitscont a        */
                                   /* multiple of 64 ?             */
if((multiple=(uncmprsdbitscont>>6))>0)
        {
                                   /* Compress the multiple of     */
        bitcont=multiple+63;       /* 64 part.                     */
        code=FAX[color][bitcont].bits;
        length=FAX[color][bitcont].length;
        cmprscounter=cmprscounter+length;
                                   /* Is  old bitsleft > length ?  */
        if ((bitsleft=bitsleft-length)>0)
                                   /* Put the new code at the      */
                                   /* current compressed word,     */
                                   /* using the new bitsleft to put*/
                                   /* it in the correct position . */
                (*cmprsdwordptr)|=code<<(bitsleft);
        else                       /* The old bitsleft <= length.  */
                {                  /* Negate bitsleft and put part */
                                   /* of the code that fills the   */
```

```
                                   /* word in the compressed word. */
                (*cmprsdwordptr)|=(code) >> (-bitsleft);
                                   /* Move to a new  word  and put */
                                   /* the rest of the code in a    */
                                   /* new compressed word, filling */
                                   /* from the left to the right.  */
                *(++cmprsdwordptr)=(code) <<
                                      (bitsleft = (16 + bitsleft));
                }
                                   /* Now compress the part that   */
                                   /* is less than 64 bits.        */
                                   /* If the no. of bits = 640 we  */
        if(multiple<10)            /* skip putting the  zero part. */
                {
                                   /* bitcont is the remainder of  */
                                   /* dividing uncmprsdbitscont by */
                                   /* 64.                          */
                bitcont=uncmprsdbitscont & maskl;
                                   /* Get the corresponding  code  */
                                   /* and the code-length.         */
                code=FAX[color][bitcont].bits;
                length=FAX[color][bitcont].length;
                                    /* Update cmprscounter by  the */
                                    /* code-length.                */
                cmprscounter=cmprscounter+length;
                                   /* If  there  are still  more   */
                                   /* unprocessed  bits  in  the   */
                                   /* current  word  then  put the */
                                   /* compressed  bits  in  the    */
                                   /* corresponding part of the    */
                                   /* word in the compressed buffer*/
                if(( bitsleft=bitsleft-length)>0 )
                        (*cmprsdwordptr)|=code<<(bitsleft);
                else
                        {          /* Otherwise  split  the  code  */
                                   /* among  the  current and the  */
                                   /* next words of the compressed */
                                   /* buffer.
                        ((*cmprsdwordptr))|=(code) >> (-bitsleft);
                        (*++cmprsdwordptr)=(code) <<
                                     (bitsleft = (16 + bitsleft));
                        }
                }
        }
                                   /* Run-length  was  less  than  */
else                               /* 64 bits.                     */
        {
                                   /* Get the corresponding number */
                                   /* of  bits  and  run-length    */
                                   /* then update "cmprscounter".  */
```

```
        code=FAX[color][uncmprsdbitscont].bits;
        length=FAX[color][uncmprsdbitscont].length;
        cmprscounter=cmprscounter+length;
                                   /* Same case as before.         */
        if ((bitsleft=bitsleft-length)>0)
                (*cmprsdwordptr)|=code<<(bitsleft);
        else
                {
                ((*cmprsdwordptr))|=(code) >> (-bitsleft);
                (*++cmprsdwordptr)=(code) <<
                                 (bitsleft = (16 + bitsleft));
                }
        }
}
/*---------------------- UPDATE_CMPRSDBLK() -----------------------*/

/*====================== INIT_CMPRSDBLK() ==========================*/
/* Initializes the compression buffer pointer to the first word of */
/* space allocated, sets the compression counter to zero and starts*/
/* with the most left bit of the first word in the compressed      */
/* buffer.
/*==================================================================*/
void    init_cmprsdblk(newblkptr)
unsigned        *newblkptr;

{
        cmprsdwordptr=newblkptr;
        bitsleft=16;
        cmprscounter=0;
}
/*--------------------- END INIT_CMPRSDBLK() ----------------------*/

/*===================== updt_cmprsblk_code ========================*/
/* Updates the compression buffer 'cmprsblk' by going to the next  */
/* code after the passed 'code' with 'length' of bits.             */
/*==================================================================*/
void    updt_cmprsblk_code(code,length)
register        unsigned        code;
register        int             length;

{
cmprscounter=cmprscounter+length;  /* Update "cmprscounter".       */
if ((bitsleft=bitsleft-length)>0)  /* If old bitsleft > length,    */
                                   /* then put the new code at the */
                                   /* current cmprsdword, using a  */
                                   /* new bitsleft.                */
        (*cmprsdwordptr)|=code<<(bitsleft);
else                               /* Old bitsleft <= length.      */
        {                          /* Negate bitsleft and put part */
                                   /* of the code that fills the   */
```

```
                                   /* word in the "word".          */
        (*cmprsdwordptr)|=(code) >> (-bitsleft);
                                   /* Move to a new word and put   */
                                   /* the   rest   of  the   code, */
                                   /* filling  from the  left.     */
        *(++cmprsdwordptr)=(code) << (bitsleft=(16 + bitsleft));
        }
}
/*-------------------- End updt_cmprsblk_code ---------------------*/

/*==================== get_cmprs_reslt() ==========================*/
/* This function returns the number of compressed bits since the   */
/* last initialization of cmprscounter.                            */
/*=================================================================*/
unsigned        get_cmprs_reslt()

{
return(cmprscounter);
}
/*----------------- END get_cmprs_reslt() ------------------------*/

/*================= set_cmprscontr_to_zero() =====================*/
/* Set_cmprscontr_to_zero() :it sets cmprscounter to zero. Uses it */
/* if you are compressing a block and want to get cmprscounter for */
/* each line alone.                                                */
/*=================================================================*/
void    set_cmprscontr_to_zero()
{
cmprscounter=0;
}
/*----------------- END set_cmprscontr_to_zero() -----------------*/
/* ----------------------- END cupdt.c -------------------------- */
```

14.4. File Dcmprs2d.c

```
#include        <memory.h>
#include        <malloc.h>
#define         findtime(tl)     {if(tend>tstart) tl=tend-tstart;\
                                  else tl=6000-tstart+tend;}
#define         update_dcmprs_code(lcolor,llength)              \
                            {{if(lcolor)                         \
                                 update_dcmprs_whitereg(llength);\
                             else                                \
                                 update_dcmprs_blakreg(llength);}}
#define         switchcolor      {tmpcolor=a0color;\
                                  a0color=a1color; \
                                  a1color=tmpcolor;}
#define         KFACTOR           2
#define         BLACKCHAR        '0'
```

```
#define         WHITECHAR       '1'
#define         BLACK           0
#define         WHITE           1
/*
 * prvslinestart : Points to the head of the previous line.
 * currentword   : Current word of the cmprsdbufr.
 * dcmprstime    : Decompression time.
 * xsize, ysize  : Horizontal and vertical dimensions, in bits,
 *                 of the screen block.
 * xmaxplsl      : xsize + 1.
 * ymaxplsl      : ysize + 1.
 * xsizeinbytes  : Horizontal dimension, in bytes,
 *                 of the screen block.
 */
char                    *prvslinestart;
static  unsigned        dcmprstime,ysize,ymaxplsl;
static  unsigned        xsize,xmaxplsl,xsizeinbytes;
unsigned                currentword;

/* ==================== dcmprs_blk_2d ======================= */
/* In this function the first line is One-Dimensionally decoded.   */
/* The reference line is set to point to that line, then the       */
/* following k-1 lines are Two-Dimensionally decoded with respect  */
/* to the reference line which is updated to point to the previous */
/* line every time a line is decoded.                              */
/* ============================================================== */
void    dcmprs_blk_2d()

{
register        int     i,k;
unsigned                tstart,tend;
char                    *refrenceline;

                                /* The reference line is the    */
                                /* line just before the coding  */
refrenceline=malloc(xmaxplsl+1); /* line.                       */
tstart=gttime();
                                /* Pointer to the previous line.*/
                                /* It  is  updated  at  the     */
                                /* beginning  of  the   loop    */
                                /* and  thus it  will be set to */
                                /* point to an  imaginary line  */
                                /* before the  first  line  in  */
prvslinestart -=xsizeinbytes;   /* the screen.                  */
                                /* Loop  until all  lines  are  */
                                /* processed.  The  first  line */
for(i=1; i < ymaxplsl; )        /* of k lines is 1-d decoded.   */
        {
        dcmprs_line_1d();       /* One-dimensional  decoding.   */
        i++;
```

```
        k = KFACTOR-1;
                                   /* Point to the previous line.  */
        prvslinestart +=xsizeinbytes;
                                   /* Decode k-1 lines, after the  */
                                   /* previously 1d decoded line,  */
                                   /* the using  Two-Dimensional   */
                                   /* decoding algorithm.          */
        while( k-- && i < ymaxplsl )
                {
                swapbits_to_string(prvslinestart,
                                        refrenceline+1, xsize);
                                   /* Two-Dimensional decoding.    */
                dcmprs_line_2d(refrenceline);
                                   /* Point to the previous line.  */
                prvslinestart +=xsizeinbytes;
                i++;
                }
        }
tend=gttime();
findtime(dcmprstime)               /* ------------ MACRO ----------*/
free(refrenceline);                /* Free allocated memory.       */
}
/* -------------------- END dcmprs_blk_2d ------------------------ */

/* ====================== dcmprs_line_2d ======================== */
/* With respect to the reference line ( previous line ) the current*/
/* line is decoded. The relative positions of a0, a1, a2, on the   */
/* coding line, and b1, b2, on the reference line, determine       */
/* whether the decoding mode is the pass, horizontal or vertical   */
/* mode. The decoded line is updated as each mode is realized until*/
/* the end of line is reached.                                     */
/* Before updating the decompression buffer with the run of bits we*/
/* must note the following point: Since a0, at the start of every  */
/* line, was set to an imaginary black changing element, then the  */
/* first black run length should not count this imaginary pel.     */
/* ============================================================== */
dcmprs_line_2d(refrenceline)
char    *refrenceline;

{
register unsigned a0;
unsigned          a1,a2,a0a1,a1a2;
unsigned          b1,b2;
int               a0color,a1color,tmpcolor;
char              *tmpptr;
static   int      blackbits,wtbits;
static   int      *blackbitsptr=&blackbits,*wtbitsptr=&wtbits;

a0=0;                              /* First pixel in the decoding  */
                                   /* line.                        */
```

```
refrenceline[a0] = BLACKCHAR ;
a0color=BLACK;
a1color=WHITE;
while( a0 < xmaxpls1 )
{                                  /* Refer to the comments in file*/
                                   /* cmprs2d.c  for  explanation  */
                                   /* about the code and how to    */
                                   /* detect a1, a2, b1, and b2.   */

                                   /*     Detect  b1.              */
   if( refrenceline[a0] == (a1color+BLACKCHAR) )
   {
       if(tmpptr=memchr(&refrenceline[a0+1], a0color+BLACKCHAR,
                                                    xmaxpls1-a0))
      {
            b1=tmpptr-refrenceline;
            if(tmpptr=memchr(tmpptr+1,a1color+BLACKCHAR,xmaxpls1-b1))
                   b1=tmpptr-refrenceline;
            else
                   b1=xmaxpls1;
       }
       else
            b1=xmaxpls1;
   }
   else
   {
       if(tmpptr=memchr(&refrenceline[a0+1],a1color+BLACKCHAR,
                                                    xmaxpls1-a0))
              b1=tmpptr-refrenceline;
       else
              b1=xmaxpls1;
   }
                                   /*     Detect  b2.              */
   if(tmpptr=memchr(&refrenceline[b1+1],a0color+BLACKCHAR,
                                                    xmaxpls1-b1))
        b2=tmpptr-refrenceline;
   else
        b2=xmaxpls1;
   if( currentword & 0x8000 )      /* Get "bit1" of "currentword". */
      {                            /*    Vertical mode(0).         */
      if( a0==0 )                  /* Update the decompression     */
                                   /* buffer.                      */
          update_dcmprs_code(a0color,b1-(a0+1))
      else
          update_dcmprs_code(a0color,b1-a0)
      a0=b1;
      switchcolor
      update_cmprs(1);             /*   Codeword  =  1.            */
      }
   else
```

```
{
if( currentword & 0x4000 )   /*  Bit1 = 0,    get bit2.      */
     {                       /*  Bit1,2 = 01, get bit3.      */
     if( currentword & 0x2000 )
        {                    /* Vertical mode(1). a1 to the  */
        if( a0==0 )          /* right of b1 by 1 bit.        */
                  update_dcmprs_code(a0color,b1+1-(a0+1))
        else
                  update_dcmprs_code(a0color,b1+1-a0)
        a0=b1+1;
        switchcolor
        update_cmprs(3);     /*   Codeword  = 011.           */
        }
     else
        {                    /* Vertical mode(-1). a1 to the */
        if(a0==0)            /* left of b1 by 1 bit.         */
                  update_dcmprs_code(a0color,b1-1-(a0+1))
        else
                  update_dcmprs_code(a0color,b1-1-a0)
        a0=b1-1;
        switchcolor
        update_cmprs(3);     /*    Codeword  = 010.          */
        }
     }
 else
     {                       /* Bit1,2 = 00, get bit3.       */
     if( currentword & 0x2000 )
        {                    /*  Horizontal mode.            */
        update_cmprs(3);     /*  Codeword  =  001.           */

                             /* Decode  the  following  two  */
                             /* codes using One-Dimensional  */
                             /* decoding  scheme according   */
        if( a0color )        /* to the a0 color.             */
            {                /* White  code  followed  by a  */
                             /* black one.                   */
            uncmprs_white(wtbitsptr);
            uncmprs_blak(blackbitsptr);
            }
        else
            {                /* Black  code  followed  by a  */
                             /* white one.                   */
            uncmprs_blak(blackbitsptr);
            uncmprs_white(wtbitsptr);
            }
        if( a0==0 ) blackbits++;
                             /* Bypass  the last two hori-   */
                             /* zontal codes.                */
            a0 += blackbits + wtbits;
        }
```

```
else
   {                    /* Bit1,2,3 = 000, get bit4.     */
   if( currentword & 0x1000 )
       {                /* Pass mode. Codeword=0001.     */
       if(a0==0)
            update_dcmprs_code(a0color,b2-(a0+1))
       else
            update_dcmprs_code(a0color,b2-a0)
       a0=b2;
       update_cmprs(4); /* Update the buffer with 4 bits.*/
       }
   else
       {
                        /* Bit1,2,3,4=0000, get bit5.    */
       if( currentword & 0x0800 )
            {
                        /* Bit1,2,3,4,5 = 00001,         */
                        /* get bit6.                     */
            if( currentword & 0x0400 )
                 {      /* Vertical mode(2). a1 to the  */
                        /* right of b1 by 2 bits.        */
                 if(a0==0)
                      update_dcmprs_code(a0color,
                                             b1+2-(a0+1))
                 else
                      update_dcmprs_code(a0color,b1+2-a0)
                      a0=b1+2;
                      switchcolor
                        /*   Codeword = 000011.          */
                      update_cmprs(6);
                 }
            else
                 {      /* Vertical mode(-2). a1 to the */
                        /* left  of b1 by 2 bits.        */
                 if(a0==0)
                      update_dcmprs_code(a0color,
                                             b1-2-(a0+1))
                 else
                      update_dcmprs_code(a0color,b1-2-a0)
                      a0=b1-2;
                      switchcolor
                        /*   Codeword = 000010.          */
                      update_cmprs(6);
                 }
            }
       else
           {            /* Bit1,2,3,4,5 = 00000,         */
                        /* get bit6.                     */
           if( currentword & 0x0400 )
                {       /* Bit1,2,3,4,5,6 = 000001,      */
```

```
                                   /* get bit6.                     */
                            if( currentword & 0x0200 )
                                 { /* Vertical mode(3). a1 to  the */
                                   /* right of b1 by 3 bits.       */
                                 if(a0==0)
                                      update_dcmprs_code(a0color,
                                                        b1+3-(a0+1))
                                 else
                                      update_dcmprs_code(a0color,
                                                        b1+3-a0)
                                 a0=b1+3;
                                 switchcolor
                                   /*   Codeword = 0000011.        */
                                 update_cmprs(7);
                                 }
                            else
                                 { /* Vertical mode(-3). a1 to the */
                                   /* left  of b1 by 3 bits.       */
                                 if(a0==0)
                                      update_dcmprs_code(a0color,
                                                        b1-3-(a0+1))
                                 else
                                      update_dcmprs_code(a0color,
                                                        b1-3-a0)
                                 a0=b1-3;
                                 switchcolor
                                   /*    Codeword = 0000010.       */
                                 update_cmprs(7);
                                 }
                            }
                       else
                            {      /* Bit pattern = 000000 should  */
                                   /* never happen unless there are*/
                                   /* some errors.                 */
                            printf("extra code   \n");
                            exit();
                            }
                       }
                  }
              }
           }
       }
    }
}
/* --------------------- END dcmprs_line_2d --------------------- */

/* ================== init_dcmprs_blk_2d ====================== */
/* Initialize local variables to this file.                      */
/* ============================================================= */
void    init_dcmprs_blk_2d(xsizein,ysizein,dcmprsbuffere)
```

```
unsigned        xsizein,ysizein;
char            *dcmprsbuffere;
{
ysize=ysizein;
xmaxplsl=xsizein+1;
xsizeinbytes=(xsizein/8)+(xsizein%8>0);
prvslinestart=dcmprsbuffere;
xsize=xsizein;
ymaxplsl=ysizein+1;
}
/* ------------------- END init_dcmprs_blk_2d -------------------- */

                                /* Refer to the comments in file  */
                                /* dupdtc.c in appendix B section */
                                /* 13.6 for all the coming code.  */
static  unsigned        currentword;
static  unsigned        nextword,*nextwordptr;
static  unsigned        cbitsremain;
static  unsigned        rightbitsword[]={0,0x0001,0x0003,0x0007,
                                  0x000f,0x001f,0x003f,
                                  0x007f,0x00ff,0x01ff,
                                  0x03ff,0x07ff,0x0fff,
                                  0x1fff,0x3fff,0x7fff,
                                  0xffff};
unsigned                leftbitsword []={0,0x8000,0xc000,0xe000,
                                  0xf000,0xf800,0xfc00,
                                  0xfe00,0xff00,0xff80,
                                  0xffc0,0xffe0,0xfff0,
                                  0xfff8,0xfffc,0xfffe,
                                  0xffff};

/*====================== UPATE_CMPRS() ========================*/
/* This function updates "currentword", which is a window into the */
/* compressed buffer.                                              */
/*==============================================================*/
update_cmprs(codelngth)
int     codelngth;

{
register        unsigned        tempword;
register        int             difference;

tempword = currentword;
tempword <<= codelngth;
if((difference = cbitsremain-codelngth) > 0)
        tempword |= nextword>>(difference);
else
        {
        difference =- difference;
        tempword |= nextword << (difference);
```

```
        nextword = *(++nextwordptr);
        tempword |= nextword >> (difference=(16- (difference)) );
        }
nextword &= rightbitsword[difference];
cbitsremain = difference;          /* Update  cbitsremain.         */
currentword = tempword;            /* Update current word.         */
return( tempword );
}
/*--------------------- END UPDATE_CMPRS ---------------------------*/

/*======================== init_cmprs =============================*/
init_cmprs(cmprsbfrptr)
unsigned       *cmprsbfrptr;

{
cbitsremain = 16;
currentword = *(cmprsbfrptr);
nextword    = *(nextwordptr=cmprsbfrptr+1);
}
/*--------------------- End init_cmprs ----------------------------*/

/*======================= MATCH_BLAK ==============================*/
match_blak(clrbitsptr,codebitsptr)
register        int    *clrbitsptr;
int                    *codebitsptr;

{
static unsigned        BLK_CODES[] =
                       {
                                   /* BARRAY_4 bits.               */
                                0x7000,0x8000,0xb000,0xc000,0xe000,
                                0xf000,
                                   /* BARRAY_5 bits.               */
                                0x9800,0xa000,0x3800,0x4000,0xd800,
                                0x9000,
                                   /* BARRAY_6 bits.               */
                                0x1c00,0x2000,0x0c00,0xd000,0xd400,
                                0xa800,0xac00,0x5c00,
                                   /* BARRAY_7 bits.               */
                                0x4e00,0x1800,0x1000,0x2e00,0x0600,
                                0x0800,0x5000,0x5600,0x2600,0x4800,
                                0x3000,0x6e00,
                                   /* BARRAY_8 bits.               */
                                0x3500,0x0200,0x0300,0x1a00,0x1b00,
                                0x1200,0x1300,0x1400,0x1500,0x1600,
                                0x1700,0x2800,0x2900,0x2a00,0x2b00,
                                0x2c00,0x2d00,0x0400,0x0500,0x0a00,
                                0x0b00,0x5200,0x5300,0x5400,0x5500,
                                0x2400,0x2500,0x5800,0x5900,0x5a00,
                                0x5b00,0x4a00,0x4b00,0x3200,0x3300,
```

```
                                0x3400,0x3600,0x3700,0x6400,0x6500,
                                0x6800,0x6700
                        };
static int      BLK_RUNS[] =
                        {          /*  BCODE_4 bits.               */
                                2  ,3  ,4  ,5   ,6   ,7  ,
                                   /*  BCODE_5 bits.               */
                                8   ,9   ,10   ,11  ,-64  ,-128 ,
                                   /*  BCODE_6 bits.               */
                                1, 12, 13, 14, 15, 16, 17, -192  ,
                                   /*  BCODE_7 bits.               */
                                18, 19, 20, 21, 22, 23, 24, 25, 26,
                                27, 28, -256,
                                   /*  BCODE_8 bits.               */
                                0, 29, 30, 31, 32, 33, 34, 35, 36,
                                37 , 38, 39, 40, 41, 42, 43, 44, 45,
                                46, 47, 48, 49, 50, 51, 52, 53, 54,
                                55, 56, 57, 58, 59, 60, 61, 62, 63,
                                -320, -384, -448, -512, -576, 640
                        };
static  int  BGROUPS[]={5,   4,6,  5,6,  6,8,  7,12,  8,42 };
register        word;

word = currentword;
switch (1)
   {
        case 1:
        {
        if( match_all_bits(word,BLK_CODES,BLK_RUNS,BGROUPS,
                                clrbitsptr,codebitsptr) )
                break;
        }
        default : {
                  printf("Wrong code encountered in 'match_blak'\n");
                  exit(0) ;
                  }
     }
}
/*------------------------ END MATCH_BLAK --------------------------*/

/*======================= MATCH_WHITE =============================*/
match_white(clrbitsptr,codebitsptr)
int            *clrbitsptr,*codebitsptr;

{
                                   /* See the comment for BLK_CODES*/
static  unsigned        WHITE_CODES[] =
                        {
                                   /* Codebits = 10.               */
                                0x05c0, 0x0600, 0x0200, 0x03c0,
```

```
                                0x0dc0,
                                   /* WARRAY_11 bits.              */
                                0x0ce0, 0x0d00, 0x0d80, 0x06e0,
                                0x0500, 0x02e0, 0x0300,
                                   /* WARRAY_12 bits.              */
                                0x0ca0, 0x0cb0, 0x0cc0, 0x0cd0,
                                0x0680, 0x0690, 0x06a0, 0x06b0,
                                0x0d20, 0x0d30, 0x0d50, 0x0d60,
                                0x0d70, 0x06c0, 0x06d0, 0x0da0,
                                0x0db0, 0x0540, 0x0550, 0x0560,
                                0x0570, 0x0640, 0x0650, 0x0520,
                                0x0530, 0x0240, 0x0370, 0x0380,
                                0x0270, 0x0280, 0x0580, 0x0590,
                                0x02b0, 0x02c0, 0x05a0, 0x0660,
                                0x0670, 0x0c80, 0x0c90, 0x05b0,
                                0x0330, 0x0340, 0x0350,
                                   /* WARRAY_13 bits.              */
                                0x0360, 0x0368, 0x0250
                        };

                                   /* See the comment for BLK_RUNS.*/
static  int     WHITE_RUNS[] =
                        {
                                   /* WCODE_10 BITS.               */
                                16, 17, 18, -64, 0,
                                   /* WCODE_11 bits.               */
                                19, 20, 21, 22, 23, 24, 25,
                                   /* WCODE_12 bits.               */
                                26, 27, 28, 29, 30, 31, 32, 33,
                                34, 35, 36, 37, 38, 39, 40, 41,
                                42, 43, 44, 45, 46, 47, 48, 49,
                                50, 51, 52, 53, 54, 55, 56, 57,
                                58, 59, 60, 61, 62, 63, -128, -192,
                                -256, -320, -384, -448,
                                   /* WCODE_13 bits.               */
                                -512, -576, 640
                        };
static  unsigned  WGROUPS[]={4, 10,5,   11,7,   12,44,  13,3 };
register        unsigned        tmpword,word;

word = currentword;
switch (1)
   {
        case 1:
        {
        if(word & 0x8000)               /*  Bit 16 = 1.            */
                {
                if(word & 0x4000)       /* Bit 15 = 1 then code=2. */
                        *clrbitsptr = 2;
                else                    /*     Bit 15 = 0.         */
```

```
                *clrbitsptr = 3;
        *codebitsptr = 2;       /*  Code length = 2.        */
        break;
        }
if(word & 0x4000)               /*    Bit 15 = 1.           */
        {
        if(word & 0x2000)       /*    Bit 14 = 1.           */
                *clrbitsptr=4;  /*    Code = 4 .            */
        else                    /*    Bit 14 = 0.           */
                *clrbitsptr=1;  /*    Code = 4.             */
        *codebitsptr=3;         /*    Code length = 3.      */
        break;
        }
if(word & 0x2000)               /*    Bit 14 = 1.           */
        {
        if(word & 0x1000)       /*    Bit 13 = 1.           */
                *clrbitsptr=5;  /*    Code = 5.             */
        else                    /*    Bit 13 = 0.           */
                *clrbitsptr=6;  /*    Code = 6.             */
        *codebitsptr=4;         /*    Code length = 4.      */
        break;
        }
if(word & 0x1000)               /*    Bit 13 = 1.           */
        {
        if(word & 0x0800)       /*    Bit 12 = 1.           */
                {
                *clrbitsptr=7;  /*    Code = 7.             */
                *codebitsptr=5; /*    Code length = 5.      */
                break;
                }
        else                    /*    Bit 12 = 0.           */
                {               /*    Bit 11 = 1.           */
                if(word & 0x0400)
                                /*    Code = 8.             */
                        *clrbitsptr=8;
                else            /*    Bit 11 = 0.           */
                                /*    Code = 9.             */
                        *clrbitsptr=9;
                *codebitsptr=6; /*    Code length = 6.      */
                break;
                }
        }
if((tmpword=(word&0xfe00)) == 0x0800)
        { *codebitsptr=7;       *clrbitsptr=10; break;  }
if(tmpword==0x0a00)
        { *codebitsptr=7;       *clrbitsptr=11; break;  }
if(tmpword==0x0e00)
        { *codebitsptr=7;       *clrbitsptr=12; break;  }
if((tmpword=(word&0xff00)) == 0x0400)
        { *codebitsptr=8;       *clrbitsptr=13; break;  }
```

```
        if(tmpword==0x0700)
                { *codebitsptr=8;       *clrbitsptr=14; break;  }
        if((word&0xff80)==0x0c00)
                { *clrbitsptr=15;       *codebitsptr=9; break;  }
        if( match_all_bits(word,WHITE_CODES,WHITE_RUNS,WGROUPS,
                                clrbitsptr,codebitsptr) )
                break;
        }
        default : {
                  printf(
                    " Wrong code encountered in 'match_white'\n");
                  exit(0) ;
                  }
   }
}
/*--------------------- END MATCH_WHITE ---------------------------*/

/* ===================  get_dcmprstime() ========================== */
unsigned        get_dcmprstime()
{
return(dcmprstime);
}
/* ------------------- END get_dcmprstime() ---------------------- */
/* ---------------------- END dcmprs2d.c ------------------------- */
```

14.5. File Dcmprsln.c

```
#include        <stdio.h>
#include        <io.h>
#include        "colordef.h"

int     update_cmprs(int);
int     uncmprs_blak(int *),            uncmprs_white(int *);
int     match_blak(int *,int *),        match_white(int *,int *);
int     update_dcmprs_blakmk(int),      update_dcmprs_whitemk(int);
int     update_dcmprs_blakreg(int),     update_dcmprs_whitereg(int);

/*
 *  Refer to the file "dcmprsln.c" in appendix B section 13.5 for
 *  comments.
 */

/*========================= DCMPRSLN() ============================*/
dcmprs_line_1d()

{
int                     clrbits;
register        int     *clrbitsptr=&clrbits;
```

```
while( uncmprs_blak(clrbitsptr) && uncmprs_white(clrbitsptr) )
                                     ;
}
/*------------------------ END DCMPRSLN() -------------------------*/

/*======================== UNCMPRS_BLAK() ========================*/
uncmprs_blak(nmbrblackbitsptr)
int     *nmbrblackbitsptr;

{
int                             clrbits,codebits;
register        int             *clrbitsptr=&clrbits;
register        int             *codebitsptr=&codebits;

*nmbrblackbitsptr = 0;
match_blak(clrbitsptr,codebitsptr);

if(*clrbitsptr<0)
        {
        *clrbitsptr=-*clrbitsptr;
        *nmbrblackbitsptr += clrbits;
        update_cmprs(*codebitsptr);
        update_dcmprs_blakmk(*clrbitsptr);
        match_blak(clrbitsptr,codebitsptr);
        }
update_cmprs(*codebitsptr);
*nmbrblackbitsptr += clrbits;
return( update_dcmprs_blakreg(*clrbitsptr));
}
/*---------------------- END UNCMPRS_BLK() ------------------------*/

/*====================== UNCMPRS_WHITE() =========================*/
uncmprs_white(nmbrwhitebitsptr)
int     *nmbrwhitebitsptr;

{
int                             clrbits,codebits;
register        int             *clrbitsptr=&clrbits;
register        int             *codebitsptr=&codebits;

*nmbrwhitebitsptr = 0;
match_white(clrbitsptr,codebitsptr);
if(*clrbitsptr<0)
        {
        *clrbitsptr=-*clrbitsptr;
        *nmbrwhitebitsptr += clrbits ;
        update_cmprs(*codebitsptr);
        update_dcmprs_whitemk(*clrbitsptr);
        match_white(clrbitsptr,codebitsptr);
        }
```

```
update_cmprs(*codebitsptr);
*nmbrwhitebitsptr += clrbits ;
return( update_dcmprs_whitereg(*clrbitsptr));
}
/*--------------------- END UNCMPRS_WHITE() -----------------------*/
/* --------------------- END dcmprsln.c ------------------------ */
```

14.6 File Bitsrng.asm

```
NAME            bitsrng
TITLE           SWAP BYTES THEN CONVERT BITS TO STRING
PUBLIC          _swapbits_to_string
DGROUP  GROUP   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
LASTBITS        EQU     [BP-2]
WORDCONT        EQU     [BP-4]
EXTRN           __chkstk:NEAR
_TEXT           SEGMENT BYTE PUBLIC  'CODE'

_swapbits_to_string      PROC   NEAR
         PUSH   BP
         MOV    BP,SP
         MOV    AX,4
         CALL   __chkstk
         PUSH   DI
         PUSH   SI
         PUSH   ES
         PUSH   DS
         POP    ES
         MOV    SI,[BP+4]
         MOV    DI,[BP+6]
         MOV    AX,[BP+8]
         MOV    DX,AX
         MOV    CX,4
         SHR    DX,CL
         MOV    WORDCONT,DX
         AND    AX,000FH
         MOV    LASTBITS,AX
LOOP1:   MOV    CX,16
         LODSW
         XCHG   AH,AL
         MOV    DX,AX
         MOV    BX,8000H
LOOP2:   TEST   DX,BX
         JZ     ZERO_BIT
ONE_BIT: MOV    AX,'1'
         STOSB
         JMP    SHIFT_MASK
```

```
ZERO_BIT:
         MOV     AX,'0'
         STOSB
SHIFT_MASK:
         SHR     BX,1
         LOOP    LOOP2
         DEC     WORD PTR WORDCONT
         JNZ     LOOP1
LAST_BITS:
         CMP     BYTE PTR LASTBITS,0
         JZ      BITSTRING_CODE
         MOV     CX,LASTBITS
         LODSW
         XCHG    AH,AL
         MOV     DX,AX
         MOV     BX,8000H
LOOP3:   TEST    DX,BX
         JZ      ZERO_BIT_L
ONE_BIT_L:
         MOV     AX,'1'
         STOSB
         JMP     SHIFT_MASK_L
ZERO_BIT_L:
         MOV     AX,'0'
         STOSB
SHIFT_MASK_L:
         SHR     BX,1
         LOOP    LOOP3
BITSTRING_DONE:
         POP     ES
         POP     SI
         POP     DI
         MOV     SP,BP
         POP     BP
         RET
_swapbits_to_string     ENDP
_TEXT            ENDS
END
/* ---------------------- END bitsrng.asm ---------------------- */
```

15. APPENDIX D. PROGRAM LIST OF METHOD LZW

The C programs in this appendix and the following appendices use the function "Indx" from C Power Packs by Software Horizons Inc.

The files in this listing make use of the files in the following sections:

- Appendix B: 13.9, 13.11, and 13.12.

15.1. File Main.c

```
/*==============================================================*/
/* This program simulates the Lempel-Ziv-Welch approach to compress*/
/* data and then decompress it according to the same approach.   */
/* This alogrithm is adaptive in the sense that it starts with an */
/* empty table of symbol strings and builds the table during both */
/* the compression and decompression processes. These are one-pass */
/* procedures that require no prior information about the input  */
/* data statistics and execute in time proportional to the length */
/* of the message.                                                */
/*==============================================================*/

#include        <stdio.h>
#include        <memory.h>
#include        <dos.h>
#include        <io.h>
#include        <fcntl.h>
#include        <malloc.h>
#include        <sys\types.h>
#include        <sys\stat.h>
#include        <string.h>

#define         LINT_ARGS
#define         HI_RES          6       /* 640x200 graphics mode.  */
#define         TEXT_MODE       3       /* Text mode.              */
#define         ALPHABET_SIZE   256     /* Sizes of alphabet and   */
#define         MAX_SIZE        4096    /* code tables.            */
#define         SCRN_SIZE       16004
#define         uchar           unsigned char
#define         findtime(time)  { tend=gttime();\
                                 if(tend>tstart)  time=tend-tstart;\
                                 else  time=(6000-tstart)+tend;}

/* Declare variables :                                             */
/* Strings table consists of two parts, the first one is of  word  */
/* type while the other one is of character type. This is due to   */
/* the fact that only 20 bits are needed to represent each string  */
/* so no more than 3 bytes are needed for this representation.     */

static          char    far     data_bufr[32000] ;
static          char            work_bufr[SCRN_SIZE] ;
```

```
                                /* Window coordinates.             */
int             x1=0,y1=0,x2=639,y2=199 ;
char            datafile[41];
unsigned        bufr_size ;     /* Holds the screen size in bytes. */

unsigned        gttime();
void            init_screen( unsigned );
void            decompress( char *, char far *, unsigned ) ;
void            compress( char * , char far * , unsigned * ) ;

main(argc, argv)
int     argc;
char    *argv[];

{       unsigned        tstart,tend,cmprstime,dcmprstime, temp, i ;
                                /* Cmprsfactor  =  original  size  */
                                /* divided by compressed size.     */
        float           cmprsfactor;

        if( argc < 6 )          /* No data was entered at the      */
          {                     /* command line.                   */
                printf("enter x1 y1 x2 y2 \n");
                scanf("%d %d %d",&x1,&y1,&x2,&y2);
                                /* Get rid of extra charcaters.    */
                while((getchar())!='\n')
                        ;
           }
        else
          {
                x1=atoi(argv[2]); y1=atoi(argv[3]);
                x2=atoi(argv[4]); y2=atoi(argv[5]);
          }
        if( argc > 1 )
                strcpy( datafile, argv[1] );
        init_screen( argc ) ;
                                /* Store the original size.        */
        cmprsfactor = ( float )bufr_size ;
        init_table() ;          /* Initialize buffers and tables.  */

                                /* Get  the  data  in  the  screen */
                                /* memory then display it again.   */
        get( x1, y1, x2, y2, work_bufr ) ;
        for(i=0;i<=55000;i++) ; /* A delay loop.                   */
        setscmode(TEXT_MODE);
        printf(" Compression is in progress \n" ) ;
        tstart=gttime();        /* Record the start of compression.*/

                        /* Compress the data in data_bufr using LZW*/
                        /* algorithm and return the compressed data*/
                        /* in the data_bufr. The work_bufr is used */
```

```
                        /* for  internal  manipulation  within    */
                        /* compress() and other function it calls. */
                        /* The size of the  compressed  buffer  is */
                        /* returned in bufr_size.                 */
                        /* We  used  data_bufr+4  so  we  will not */
                        /* compress the x and y sizes.            */
        compress( work_bufr+4, data_bufr, &bufr_size ) ;
        findtime( cmprstime )  /* -- MACRO  to  find  cmprstime.  */
        printf(" Now decompression is in progress \n" ) ;
        init_table() ;          /* Reinitialize the tables.       */
        tstart=gttime();        /* Record start of decompression. */
                        /* Decompress  data  stored  at  address  */
                        /* data_bufr+4 and with size = bufr_size.  */
                        /* Use the work_bufr in function decompress*/
                        /* for its internal use.                  */
        decompress( work_bufr+4, data_bufr, bufr_size ) ;
        findtime( dcmprstime )  /* -- MACRO  to  find dcmprstime.  */
                                /* Display data on  the screen to  */
        setscmode(HI_RES);      /* make sure the program is working*/
        put( x1, y1, work_bufr ) ;
        for(i=0;i<=55000;i++) ; /* A delay loop.                  */
        setscmode(TEXT_MODE);
                                /* A dummy variable.              */
        passparmtrs( datafile, x1, y1, x2, y2, temp=0 ) ;
        print( cmprstime, dcmprstime,
                             cmprsfactor = cmprsfactor/bufr_size ) ;
}
/*------------------------ END main() ---------------------------*/
/*------------------------ END main.c ---------------------------*/
```

15.2. File Cmprs.c

```
#include        <memory.h>
#include        <malloc.h>
#define         uchar           unsigned        char
#define         MAX_SIZE        4096
#define         SCRN_SIZE       16004
#define         update_string_table()   \
                        {if( next_code < MAX_SIZE )\
                            {char_table[next_code] = string.k ;\
                            int_table[next_code]  = string.w ;\
                            next_code++ ;}}

extern  unsigned  int_table[] ;   /* int_table[], char_table[]     */
extern  uchar     char_table[] ;  /* and next_code are defined     */
extern  int       next_code;      /* in tables.c                   */
extern  unsigned  extracalls ;

void    adjust_output( uchar *, uchar far *, unsigned, unsigned * );
```

```
/*====================== compress() ============================*/
/* The LZW algorithm  is  organized  around  a  translation  table */
/* that maps strings of input  symbols  into a  fixed length  code.*/
/* LZW  string table contains strings that have  been  encountered */
/* previously  in the message being compressed. The input string   */
/* is  examined  serially symbol-by-symbol in one  pass  and  the  */
/* longest recognized input string is parsed off each time.        */
/*=============================================================*/
compress( compress_io,compress_work, ptr_bufr_size )
uchar          *compress_io, far *compress_work ;
unsigned       *ptr_bufr_size ;
                                    /* Compress_io contains  data   */
                                    /* needs to be compressed as an */
                                    /* input, and compressed data   */
                                    /* as output. Compress_work is  */
                                    /* used for work as a temporary */
                                    /* output of lzw compression    */
                                    /* before we pack each code word*/
                                    /* into 12 bit code. Then the   */
                                    /* function adjust_output takes */
                                    /* it as input and put the      */
                                    /* correct 12 bit codes into    */
                                    /* compress-io.                 */
{

        uchar           *input;
        unsigned        far  *output;
        char            *ptr_new_output;
        unsigned        bufr_size;
        unsigned        newsize ;
        register int    data_index=0, code;
        int             out_index=0, found, *ptr_found=&found ;
        struct  {
                uchar       k;
                unsigned    w;
                }    string ;

        input=compress_io;
        output=(unsigned far *)compress_work;
                                    /* Read the first element in    */
                                    /* the input.                   */
        string.w = input[data_index++] ;
        bufr_size=*ptr_bufr_size;   /* Find bufr_size.              */
                                    /* Loop  while  there  is  more */
                                    /* input.                       */
        while( data_index <  bufr_size )
        {
                                    /* Read  the next  element.     */
                string.k = input[data_index++];
```

```
                                   /* Function Scanw() scans  the  */
                                   /* string and returns  the code.*/
                                   /* If the passed string is found*/
                                   /* in that case found = 1,      */
                                   /* otherwise the returned value */
                                   /* of found is = 0.             */
                code = scanw( string.w, string.k, ptr_found );
                if( found )
                {                  /* wk exists in the table :     */
                                   /* wk --> w i.e. code of new w= */
                                   /* code of a location in the    */
                                   /* int_table that has w and k.  */
                        string.w = code ;
                        continue ;
                }
                else
                {                  /* wk is not in string table :  */
                                   /* string.w --> output i.e. send*/
                                   /* code of w to the output.     */
                        output[out_index++] = string.w ;
                                   /* If the  tables are not  full */
                                   /* yet then string --> string   */
                                   /* table, i.e put w and k in    */
                                   /* int_table  and  char_table   */
                                   /* respectively  at  position   */
                                   /* next_code.                   */
                        if(next_code<MAX_SIZE)
                                update_string_table()
                        else
                                extracalls++ ;
                                   /* string.k --> string.w.       */
                        string.w = ( unsigned ) string.k ;
                }
        }
                                    /* Send the last code to the   */
                                    /* output.                     */
        output[out_index] = string.w ;
                                   /* Back the output codes from a */
                                   /* string of words format to a  */
                                   /* string  of  12  bits  codes  */
                                   /* format.  The  input  to      */
                                   /* adjust_output() is compress_ */
                                   /* work. It sends the output    */
                                   /* in  the  final  form  in     */
                                   /* compress_io.                 */
        adjust_output(compress_io ,compress_work,
                                        2*(out_index +1), &newsize);
        *ptr_bufr_size= newsize;   /* Send newsize in bufr_size.   */
}
/*------------------------ END COMPRESS() -------------------------*/
```

```
/*------------------------ END cmprs.c ----------------------------*/
```

15.3. File Dcmprs.c

```
#include        <memory.h>
#include        <stdio.h>
#include        <malloc.h>
#define         uchar           unsigned char
#define         MAX_SIZE        4096
#define         ST_MAX          1000
#define         SCRN_SIZE       16004

                        /* Add the new string to the string table. */
#define         update_string_table()   \
                        {if( next_code < MAX_SIZE )\
                                {char_table[next_code]=code.k ;\
                                int_table[next_code]  = oldcode;\
                                next_code++ ;}}

                        /* Return the  value of w and k  for  the  */
                        /* passed code.                            */
#define         look_up()  \
                        {code.w = int_table[CODE] ;\
                         code.k = char_table[CODE];}

                                   /* int_table,  char_table  and  */
extern  unsigned  int_table[] ;    /* next_code  are  defined      */
extern  uchar     char_table[] ;   /* globally in tables.c         */
                                   /* index of the next code in the*/
extern  int       next_code;       /* tables not used yet.         */
                                   /* A  stack  to be  used in the */
                                   /* abnormal case for storing    */
                                   /* characters till we reach the */
                                   /* first character of the  new  */
static  char      *stack ;         /* string.                      */
                                   /* First unused element. Stack  */
static  unsigned  stack_index=0;   /* grows upward.                */

char    pop();                     /* Returns the character at the */
                                   /* top of the stack.            */
void    readjust_input( uchar far *, uchar *, unsigned, unsigned * );

/*========================== decompress() =========================*/
/* Input is in the form of 12 bits codes stored serially. We have  */
/* to readjust them to integer format so we can store them and use */
/* them in the int_table.                                          */
/* Inputsize is the size of input in bytes.                        */
/* decmprs_io = as input to decmprs() it points to compressed data */
/* decmprs_io = as ouput of decmprs() it points to decompressed    */
```

```
/*              data                                               */
/* decmprs_work = pointer to a temporary area.                     */
/*=================================================================*/
decompress(decmprs_io, decmprs_work ,inputsize)
char            *decmprs_io, far *decmprs_work;
unsigned        inputsize;

{
        unsigned        input_index=0;
        unsigned        oldcode,incode;
                                   /* The size of the compressed   */
                                   /* data, each is stored in a    */
                                   /* word, is equal to the output */
        unsigned        newsize ;  /* size of readjust_input().    */
        register        unsigned        output_index=0;
        register        unsigned        CODE ;
                                   /* Final character in the pre-  */
        char            finchar ;  /* vious String decompression.  */
        struct  {
                   char         k;
                   unsigned     w;
                } code;
        char            *temp_ptr;
        unsigned        far  *input;
        char            *databufr;

        stack = malloc( ST_MAX );  /* Allocate memory for the stack*/
                                   /* Adjust the input from 12 bits*/
                                   /* serial codes into an array of*/
                                   /* integers and put the size of */
                                   /* the array in "newsize".      */
        readjust_input(decmprs_work,decmprs_io,inputsize,&newsize);
                                   /* Find the size of the input   */
        inputsize=(newsize/2);     /* code in words.               */
        input= (unsigned far *) decmprs_work;
        databufr=decmprs_io;
                                   /* Get the first code of the    */
                                   /* input.                       */
        CODE= oldcode= input[input_index++];
        look_up()                  /*  MACRO.                      */
                                   /* Output the first character.  */
        databufr[output_index]=finchar=code.k;
                                   /* Keep looping until all codes */
                                   /* are processed.               */
        while(input_index< inputsize  )
        {                          /* Get the next input.          */
                CODE=incode=input[input_index++];
                if(CODE >= next_code)
                                   /* CODE is not defined in the   */
                {                  /* decompression table yet.     */
```

```
                        push(finchar);
                        CODE=oldcode;
                }
                                   /* Find  the  components  w  &  */
                                   /* k of CODE.                   */
                look_up()
                                   /* if w = 0 then we have a code */
                                   /* for one of the alphabets.    */
                                   /* While CODE==code(wk) separate*/
                                   /* the k & w parts of code till */
                                   /* CODE = code(k).              */
                while(code.w!=0xffff)
                {
                        push(code.k);
                        CODE=code.w;
                        look_up()
                }
                                   /* String now begins with the   */
                                   /* last k, and the rest of it.  */
                                   /* (If string longer than one   */
                                   /* k) is in the stack.          */

                                   /* Send k to the output.        */
                databufr[++output_index]=code.k;
                finchar=code.k;    /* Finchar  =  first k  of the  */
                                   /* last string.                 */
                                   /* While the stack is not empty */
                while(stack_index) /* send data to the output.     */
                        {
                        databufr[++output_index]=pop();
                        }
                update_string_table()
                oldcode=incode;
        }
}
/*------------------------ END decompress() ------------------------*/

/*============================= push() ============================*/
/* PLace an element on the stack.                                   */
/*==================================================================*/
push( item )
char    item;                      /* Data to be pushed on the     */
                                   /* stack.                       */

{
        if( stack_index >= ST_MAX )
        {
                printf( " stack overflow in push \n" ) ;
                return ;
        }
```

```
        stack[stack_index++] = item ;

}
/*--------------------------- END push() ---------------------------*/

/*============================== pop() =============================*/
/* Retrieve the top element from the stack.                         */
/*==================================================================*/
char    pop()
{
        if( --stack_index < 0 )
        {
                printf(" Stack underflow in pop \n" ) ;
                return ('\0');
        }
        return stack[stack_index];
}
/*--------------------------- END pop() ----------------------------*/
/*-------------------------- END dcmprs.c --------------------------*/
```

15.4. File Tables.c

```
#include        <stdio.h>
#include        <memory.h>
#include        <malloc.h>
#define         MAX_SIZE        4096
#define         ALPHABET_SIZE   256
#define         uchar           unsigned char
                                   /* Definition of a GLOBAL vars. */
unsigned                int_table[MAX_SIZE] ;
unsigned        char    char_table[MAX_SIZE] ;
int                     next_code ;
unsigned                *ptr_int_table=int_table;
unsigned        char    *ptr_char_table=char_table;
unsigned                extracalls=0 ;

/*========================== init_table() ==========================*/
/* This function initializes every element in int_table to a com-   */
/* bination that will  never occur. Since the code is only 12 bits */
/* long then the 16 bits used to hold these codes are to be <=     */
/* 0xfff. For this reason in this program the 0xffff code is used   */
/* to solve the above problem. It should be noted that any combin- */
/* ation > 0xfff should work correctly as well. Then the first 256 */
/* character symbols are loaded into the char_table.                */
/*==================================================================*/
init_table()

{
register        int     index ;
```

```
                                     /* Set  every  byte  in  the    */
                                     /* int_table  to  0xffff  (i.e. */
                                     /* every code word = 0xffff) so */
                                     /* that no code will match with */
                                     /* it, because actual codes are */
                                     /* only 12 bits.                */
        memset( (char *) int_table,0xff,MAX_SIZE*2);

                                     /* Set 1st 256 of char_table to */
                                     /* be the extended ASCII codes. */
        for( index=0; index < ALPHABET_SIZE; index++ )
                char_table[index] = ( short ) index ;

        next_code = ALPHABET_SIZE;
}
/*------------------------ END init_table() -----------------------*/
/*-------------------------- END tables.c -------------------------*/
```

15.5. File Scanw.asm

```
; INPUT : ( PARAMETERS PASSED BY CALLING SUBROUTINE )
;       1) CHARCODE  = CHARACTER PART OF THE CODE,i.e K.
;       2) INTCODE   = UNSIGNED INTEGER PART OF THE CODE, i.e. W.
;       3) FOUNDADRS = ADDRESS OF CODE, i.e., WHERE WE RETURN THE
;                      CODE WHICH HAS W AND K EQUAL TO INTCODE AND
;                      CHARCODE RESPECTIVELY.
;
; OUTPUT :
;       1) BOOLEAN VARIABLE "FOUND": HAS THE FOLLOWING RETURN VALUES
;                       RETURN VALUE =  1 IF A MATCH IS FOUND
;                                       0 IF NO MATCH
;       2) THE DUNCTION RETURN VALUE IS CONTAINS THE INDEX OF THE
;          FOUND CODE, IF ANY.
;
; IT NEEDS TO SHARE THE FOLLOWING WITH WHOEVER HAS THEM:
;       1) _ptr_char_table = A POINTER TO 1ST ELEMENT IN CHAR TABLE.
;       2) _ptr_int_table  = A POINTER TO 1ST ELEMENT IN INT TABLE.
;       3) next_code   = NUMBER OF FIRST FREE CODE IN CHAR TABLE.
;                      = NUMBER OF FIRST FREE CODE IN INT TABLE.
NAME     SCAN
TITLE    SCANNING OF THE ALTERNATE TABLE TO FIND A MATCH
PUBLIC   _scanw

FOUND_PTR        EQU     [BP+8]          ; PASSED PARAMETERS.
INTCODE          EQU     [BP+4]
CHARCODE         EQU     [BP+6]

DGROUP   GROUP  CONST,  _BSS,   _DATA
```

```
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP

_DATA   SEGMENT
EXTRN   _ptr_char_table:WORD
EXTRN   _ptr_int_table:WORD
EXTRN   _next_code:WORD
        PTR_next_code  DW      ?
_DATA   ENDS

_scanw  PROC    NEAR
        PUSH    BP
        MOV     BP,SP
        PUSH    DI
        PUSH    SI
        PUSH    ES
        MOV     AX,DS
        MOV     ES,AX
                                ; INITIALIZE THE REGISTERS TO THE
        MOV     AX,INTCODE      ; CORRESPONDING PARAMETERS PASSED
        MOV     DL,CHARCODE     ; FROM THE CALLING PROGRAM.

                                ; THE FOLLOWING THREE VARIABLES ARE
                                ; DEFINED SOMEWHERE ELSE.
                                ; POINTER TO THE TABLE HOLDING
                                ; ELEMENTS OF CHARACTER TYPE. THIS
                                ; TABLE HOLDS THE SECOND PART TO BE
                                ; EXAMINED IN THE SEARCH.
        MOV     SI,_ptr_char_table
                                ; TABLE USED IN THE SEARCH. IT HOLDS
                                ; THE INTEGER PART WE SCAN FOR.
        MOV     DI,_ptr_int_table
        MOV     CX,_next_code   ; NEXT NUMBER NOT USED IN TABLES YET.
LOOP1:                          ; SCAN THE WORD TABLE STARTING FROM
        REPNE   SCASW           ; DI UP TO CX ELEMENTS BIT ZERO IS
                                ; ZERO. IF ZF= 0 WE FINISHED THE SCAN
        JNE     NOMATCH         ; BEFORE ANY MATCH. SO GO TO NOMATCH.
        MOV     BX,DI           ; ZF=1 SO WE HAD A MATCH. STORE
                                ; THE LENGTH OF SCANNED WORDS IN BX.
        SUB     BX,_ptr_int_table
        SHR     BX,1            ; GET THE NUMBER OF SCANNED WORDS.
        DEC     BX              ; ADJUST LOOP STEP ( ONE MORE WORD ).
                                ; SINCE WE HAD A WORD MATCH,
        CMP     DL,[BX+SI]      ; SEE IF WE HAVE CHAR MATCH.
                                ; IF YES THEN WE HAVE A COMPLETE
        JE      MATCH           ; MATCH. SO GO TO MATCH.
                                ; CHAR DID NOT MATCH SO TRY AGAIN
                                ; AS LONG AS CX (= REMAINING CODES TO
                                ; BE SEARCHED ) NOT EQUAL TO ZERO.
                                ; IF CX REACHED ZERO BEFORE WE HAD
                                ; ANY MATCH THEN "JNE  NOMATCH" WILL
```

```
         JMP     LOOP1             ; DROP US TO NOMATCH:
NOMATCH:
         MOV     BX,FOUND_PTR      ; NO MATCH, SO STORE ZERO IN FOUND,
         MOV     WORD PTR [BX],0 ; WHICH IS ADDRESSED BY FOUND_PTR.
         JMP     SCAN_DONE         ; SCAN IS DONE.
MATCH:   MOV     BX,FOUND_PTR      ; THERE WAS A MATCH SO STORE 1 IN
         MOV     WORD PTR [BX],1 ; FOUND.
                                   ; MAKE DI = LENGTH OF SCANNED WORDS.
         SUB     DI,_ptr_int_table
         SHR     DI,1              ; MAKE DI = NUMBER OF SCANNED WORDS.
         DEC     DI                ; ADJUST LOOP STEP (ONE MORE WORD.)
                                   ; SCAN WILL RETURN AX = CODE = NUMBER
                                   ; OF WORDS SCANNED TILL WE FOUND A
                                   ; MATCH (i.e. INDEX OF THE MATCHED
         MOV     AX,DI             ; ELEMENT IN EITHER TABLE) .
SCAN_DONE:
         POP     ES
         POP     SI
         POP     DI
         MOV     SP,BP
         POP     BP
         RET
_scanw   ENDP
_TEXT    ENDS
END
/*------------------------ END scanw.asm -------------------------*/
```

15.6. File Scrinit.c

```
#include        <stdio.h>
#include        <memory.h>
#include        <dos.h>
#include        <io.h>
#include        <fcntl.h>
#include        <malloc.h>
#define  LINT_ARGS
#define  FALSE          0
#define  TRUE           1
#define  HI_RES         6
#define  TEXT_MODE      3
#define  SCREENSIZE     16384
#define  STRERR         -1        /* Sring error, not found.      */

extern   int       x1,y1,x2,y2;   /* Window coordinates.          */
extern   char      datafile[];    /* Figure input file.           */
extern   unsigned  bufr_size ;

/*====================== init-screen() ==========================*/
/* This function displays figure on screen.                       */
```

```
/*=================================================================*/
init_screen( value )
int     value;

{
char      *screenbufr;             /* Temporary buffer.            */
int       fh1,bytesread,loop=TRUE;
char      flag, c;
                                   /* "src"  is a  far  pointer    */
char      far   *src;              /* initialized to "screenbufr". */
unsigned  blksize;

        if( value <= 1 )
        {
                while(loop)
                {
                        printf("enter name of data file \n");
                        gets(datafile);
                        printf("your data file is %s \n",datafile);
                                /* Give  the  user  a chance  to   */
                                /* correct  his  mistakes.         */
                        printf("Is the given data correct (y/n)?\n");
                        flag=getchar();
                        while( (flag!='y')&&(flag!='n') &&
                               (flag!='Y')&&(flag!='N') )
                        {
                                   /* Read the end of line.        */
                                while((c=getchar()) !='\n')
                                        ;
                                printf("enter y or n ");
                                flag=getchar();
                        }
                                   /* Read the end of line.        */
                        while((c=getchar()) !='\n')
                                ;
                        if( (flag=='y')||(flag=='Y') )
                                loop=FALSE;
                }
        }
        blksize =( ( x2-x1+1 ) * (long)( y2-y1+1 ) )/8 ;
        setscmode(HI_RES);
                                /* Read data from the input file    */
                                /* into the buffer, then use this   */
                                /* data to display the figure on    */
                                /* the screen.
                                /* Both  even and  odd  banks  are */
                                /* read  separately.  If the  file */
                                /* extension is "cut" then just     */
                                /* read data into array and then    */
                                /* put it to the screen. There is   */
```

```
                        /* no need to send the data to the */
                        /* screen memory in the latter case*/

                        /* fhl = file handler of data file.*/
fhl = open(datafile,O_RDONLY|O_BINARY);
                        /* Check   if file extension = cut.*/
if( (Indx(".cut",datafile)) != STRERR )
{
                        /* Allocate 4 bytes to read x and  */
                        /* y sizes.                        */
        screenbufr=malloc(4);
                        /* Read x and y sizes from datafile*/
                        /* into screenbufr, then put values*/
                        /* into x2 and y2 respectively.    */
        bytesread=read(fhl,screenbufr,4);
        x2=*(unsigned *)screenbufr;
        y2=*(unsigned *)(screenbufr+2);
                        /* Reallocate the required size of */
                        /* memory  to  hold  the  data  in */
                        /* the input file.                 */
        screenbufr=realloc(screenbufr,
                                 blksize=4+((x2+7)/8)*(y2));
                        /* Read the data from the file.    */
        bytesread=read(fhl,screenbufr+4,blksize);
        put(x1,y1,screenbufr);
}
else
{
                        /* Do the  first  bank  (even) by  */
                        /* allocating half the total size. */
        screenbufr=malloc(SCREENSIZE/2);
        fhl = open(datafile,O_RDONLY|O_BINARY);
                        /* Read the first bank.            */
        bytesread=read(fhl,screenbufr,SCREENSIZE/2);
        src=(char far *)(screenbufr+7);
                   /* Format has the first byte of the 1st */
                   /* bank at offset 8000 of the screen    */
                   /* segment. Move the data from the file */
                   /* to that segment. Note that in the    */
                   /* screen segment the bytes starting at */
                   /* offset 8000 till (8192-7) will be    */
                   /* filled with whatever the file has.   */
                   /* This part is not from the physical   */
                   /* screen.                              */
        movedata(FP_SEG(src),FP_OFF(src),0xb800,0x0000,
                                        (SCREENSIZE/2)-7);
        bytesread=read(fhl,screenbufr,SCREENSIZE/2);
        src=(char far *)(screenbufr);
                   /* the 1st seven bytes of the 2nd half  */
                   /* of the file are a continuation of the*/
```

```
                              /* (192-7) bytes that BASIC took from    */
                              /* the screen memory and dumped it to    */
                              /* the file. So the 2nd half of the      */
                              /* screen starts after 7 bytes of the    */
                              /* 2nd part of the file. By copying the */
                              /* second half of the file into offset  */
                              /* (0x2000-7) we will fill the 7 bytes  */
                              /* at (0x2000-7) then the 2nd half of   */
                              /* the screen will be copied to offset  */
                              /* (0x2000).  This fills the odd part of*/
                              /* the screen. The remaining (192-7) of */
                              /* the file  will fill offset           */
                              /* (0x2000+8000) till offset            */
                              /* (0x2000+8000+(192-7)).               */
                movedata(FP_SEG(src),FP_OFF(src),0xb800,(0x2000-7),
                                                SCREENSIZE/2);
        }
        close(fhl);
        free(screenbufr);
        bufr_size=blksize;
}
/*------------------------- END INIT_SCREEN ----------------------*/


/*======================== SETSCMODE ===========================*/
/* sets the screen to the desired video mode.                    */
/*==============================================================*/
int setscmode(mode)                /* Function to set video mode   */
int mode;

{
union REGS inregs;
union REGS outregs;
                                   /* return the code and the      */
                                   /* interrupt for function       */
                                   /* gdosint().                   */
int ret_code,int_no;
                                   /* "set  video mode  BIOS       */
                                   /* function call.               */
   inregs.h.ah=0;
   inregs.h.al=mode;
   ret_code = int86(0x10,&inregs,&outregs);
                                   /* return the code to check for */
                                   /* any errors.                  */
   return(ret_code);
}
/*----------------------- END setscmode() ------------------------*/
/*----------------------- END Scrinit.c  ------------------------*/
```

15.7. File Print.c

```
#include        <stdio.h>
                                /* Next_code and extracalls are */
                                /* defined in tables.c.         */
extern          int             next_code ;
extern          unsigned        extracalls ;

static          unsigned        x1, y1, x2, y2, temp ;
static          char            *infile;

/*====================== passparmtrs() =======================*/
/* This function is used only for passing parameters from the main */
/* function  to this file so that they can be printed out.         */
/*==============================================================*/
passparmtrs( theinfile, c1,r1, c2,r2, dummy )
char            *theinfile;
unsigned        c1,r1,c2,r2 ;

{
        infile = theinfile  ;
        x1 = c1 ;   y1 = r1 ;
        x2 = c2 ;   y2 = r2 ;
        temp = dummy        ;
}
/*---------------------- END passparmtrs() -----------------------*/

/*======================== print() ============================*/
/* Print the results to the output.  The  data to be  printed out  */
/* are the compression time, the decompression time and the        */
/* compression factor.                                             */
/*==============================================================*/
print( cmprstime, dcmprstime, cmprsfactor )
unsigned        cmprstime, dcmprstime;
float           cmprsfactor;

{
FILE            *outfile;

printf(" Compression factor is %f \n", cmprsfactor) ;
printf(" Compression   time is %u  in 1/100 of a seconds \n",
                                                   cmprstime ) ;
printf(" decompression time is %u  in 1/100 of a seconds \n",
                                                  dcmprstime ) ;
printf(" lzw table size is %u \n",next_code);
printf(" Extra calls after tables were filled are %u \n",
                                                  extracalls ) ;

                                /* Send data to outlzw.dat file.*/
if( (outfile = fopen( "outlzw.dat", "r" )) == NULL )
```

```
        {
                                   /* Open a file for writing and  */
                                   /* then print the table heading.*/
        outfile = fopen( "outlzw.dat", "w" ) ;
        fprintf(outfile,
            "File name      x1   y1   x2   y2   cmprs     cmprs  ");
        fprintf(outfile,"dcprs   cont   table   extra \n" );
        fprintf(outfile,
            "                                   fctor     time   ");
        fprintf(outfile,"time    smbl   size    calls \n" );
        fprintf(outfile,
            "--------------------------------------------------");
        fprintf(outfile,"----------------------------\n");
        }
else
        {                          /* Append the file.             */
        outfile = fopen( "outlzw.dat", "a" ) ;
        }
                                   /* Formats of the output.       */
fprintf(outfile,
         "%-12s  %3u  %3u  %3u  %3u  %6.2f    %4u  %5u    %4u    %4u
           %4u\n", infile, x1, y1, x2, y2, cmprsfactor, cmprstime,
         dcmprstime, temp, next_code, extracalls );
}
/*-------------------------- END print() --------------------------*/
/*-------------------------- END print.c --------------------------*/
```

15.8. File Fadjst.c

```
#define         uchar   unsigned char

/*======================== adjust_output() =======================*/
/* This  procedure takes  the  compressed  output  which is in the */
/* form of words each containing 12 bits wide code from  the       */
/* procedure  compress () and packs these codes sequentially in the*/
/* output. Thus, the last 4 bits ( bits 9 thru 12 ) of  the next   */
/* code  should fit in the 4  bits at the beginning of the current */
/* word ( bits 1 thru 4 ).  This is done for every couple of words.*/
/*=================================================================*/
void    adjust_output( temp, input, oldsize, ptr_newsize )

uchar           *temp;
uchar           far *input ;
unsigned        oldsize,
                *ptr_newsize ;     /* Size of the adjusted output. */

{
        register        char            *ptr2 ;
        register        char            far *ptr1 ;
```

```
                        char            far *lastitem ;
                        unsigned        quadsize ;

                                   /* Get the even number of      */
                                   /* elements in output buffer.  */
        quadsize = ( oldsize/4 ) * 4 ;
        lastitem = input + quadsize ;
                                   /* Start adjusting the bits.   */
        for( ptr1=input, ptr2=temp; ptr1<lastitem; ptr1+=4 )
        {
                                   /* ptr1 is pointing to b3 b4 b1 */
                                   /* b2 b7 b8 b5 b6 as seen in   */
                                   /* memory, which in word form  */
                                   /* is b1 b2 b3 b4 b5 b6 b7 b8. */
                                   /* we want ptr2 to point to b2 */
                                   /* b3 b4 b6 b7 b8 = c1 c2 c3   */
                                   /* where each b represents 4   */
                                   /* bits and each c represents  */
                                   /* one byte.                   */
                *( unsigned far *) ptr1 <<= 4 ;
                                   /* *ptr1= b2 b3 b4 0 b7 b8 b5 b6*/
                                   /* *ptr1= t1 t2 t3 t4 (t=byte). */
                *ptr2++ = *(ptr1 +1) ;      /* c1 = t2.           */
                                   /* c2 = t1 bitor t3.           */
                *ptr2++ = *( ptr1 ) | *( ptr1 + 3 ) ;
                *ptr2++ = *( ptr1 + 2 ) ;   /* c3 = b7 b8.        */
        }
                                   /* If  oldsize  wasn't  evenly */
                                   /* divisible by 4 then process */
        if( oldsize - quadsize )   /* the last element in the     */
                                   /* output.                     */
                {
                *( unsigned *) ptr2 =
                            ( *( unsigned far * ) lastitem ) << 4 ;
                ptr2 +=2;          /* Adjust ptr2.                */
                }
                                   /* Return the new size of output*/
                                   /* in bytes. Ptr2 will always  */
                                   /* be pointing one byte after  */
                                   /* the last byte.              */
        *ptr_newsize =  ptr2  - temp  ;
}
/*---------------------- END adjust_output() ----------------------*/
/*------------------------ END fadjst.c ---------------------------*/
```

15.9. File Fradjst.c

```
/*======================== readjust_input() ======================*/
/* This function adjusts the form of the input data from strings of*/
```

```
/* 12 bits codes to an array of words where each word corresponds  */
/* to a 12 bit code. The left most 4 bits are set to zero. i.e.     */
/* each word = integer value of the 12 bit code.                    */
/*==================================================================*/
void    readjust_input(temporary,input,inputsize,ptr_newsize)
char            far *temporary,*input;
unsigned        inputsize,*ptr_newsize;
                                   /* Input contains the input data*/
                                   /* before this function starts. */
                                   /* It contains the adjusted data*/
                                   /* when the function is done.   */
                                   /* Inputsize= size (in bytes)   */
                                   /* of data to be adjusted.      */
                                   /* Ptr_newsize = pointer to the */
                                   /* size (in bytes) of the       */
                                   /* adjusted data.               */
{
char            char_temp;
                                   /* Number  of  the input bytes  */
unsigned        trisize;           /* divisible by 3.              */
                                   /* Points to the byte after the */
char           *lastitem;          /* trisize.                     */
register  unsigned  char  *ptr1;   /* Points to the input data.    */
register  unsigned  char  far *ptr2; /* Points to the adjusted data*/

        trisize=(inputsize/3)*3;
        lastitem=input+trisize;
                                   /* Initialize ptr1 and ptr2 to  */
                                   /* point to the input start and */
                                   /* the adjusted area start. Loop*/
                                   /* while we are inside the      */
                                   /* trisize region.              */
        for(ptr1= input, ptr2= temporary; ptr1< lastitem ;
                                        ptr1 +=3, ptr2 +=4 )
                {
                *(ptr2 +2)= *(ptr1 +2);
                *(ptr2 +3)= *(ptr1 +1) & 0x0f;
                char_temp=*ptr1;
                *(ptr1)  =*(ptr1 +1);
                *(ptr1+1)=char_temp;
                *( (unsigned far *) ptr2 )= *((unsigned *) ptr1) >>4;
                }

                                /* If inputsize was not divisible  */
                                /* by 3 then adjust the last 12    */
        if(inputsize-trisize)   /* bits (2 bytes) and store it in  */
                                /* *ptr2.                          */
                {
                *( unsigned far * )(ptr2)= (* (unsigned *) ptr1) >>4;
                ptr2 += 2 ;
```

```
                }
                                /* Newsize = size (in bytes) of    */
                                /* the readjusted  code.           */
        *ptr_newsize=(ptr2-temporary) ;
}
/*------------------------- END readjust_input() --------------------*/
/*------------------------- END fradjst.c  -------------------------*/
```

16. APPENDIX E. PROGRAM LIST OF METHOD LZWB

The files in this listing make use of the files in the following sections:

- Appendix B: 13.9, 13.11, and 13.12.
- Appendix D: 15.2 - 15.9.

16.1. File Main.c

```
#include        <stdio.h>
#include        <memory.h>
#include        <dos.h>
#include        <io.h>
#include        <fcntl.h>
#include        <malloc.h>
#include        <sys\types.h>
#include        <sys\stat.h>
#define         LINT_ARGS
#define         FALSE           0
#define         TRUE            1
#define         HI_RES          6       /* 640x200 graphics mode.  */
#define         TEXT_MODE       3       /* Text  mode.             */
#define         ALPHABET_SIZE   256     /* Size of alphabet.       */
#define         MAX_SIZE        4096    /* Table size.             */
#define         SCRN_SIZE       16004   /* 4 bytes for x & y sizes.*/
#define         uchar           unsigned char
#define         findtime(time)  { tend=gttime();\
                                 if(tend>tstart)  time=tend-tstart;\
                                 else  time=(6000-tstart)+tend;}

static          char    far     data_bufr[32000] ;
static          char            work_bufr[27000];
                                   /* Window coordinates.          */
int             x1=0, x2=639, y1=0, y2=199 ;
char            datafile[41];
unsigned        bufr_size;         /* Screen size in bytes.        */

unsigned        gttime();
unsigned        count_symbols( char *, char far *, unsigned) ;
void            decompress( unsigned  *, char far *, unsigned ) ;
void            compress( char * , char far * , unsigned * ) ;
void            dcmprs_lzw( char far * , char * , unsigned ) ;
void            swapbyts(unsigned, unsigned, unsigned ,
                                unsigned , unsigned );

main(argc, argv)
int     argc;
char    *argv[];

{
        unsigned        blksize;
```

```
unsigned        temp;
unsigned        tstart, tend, cmprstime, dcmprstime, i;
float           cmprsfactor;
char    far     *datafarptr=data_bufr ;
char    far     *workfarptr=work_bufr+4 ;

if( argc < 6 )             /* No  data  was  entered  at   */
   {                       /*the  command line.            */
        printf("enter x1 y1 x2 y2 \n");
        scanf("%d %d %d %d",&x1,&y1,&x2,&y2);
        while((getchar())!='\n')
                ;
   }
else
  {
        x1=atoi(argv[2]); y1=atoi(argv[3]);
        x2=atoi(argv[4]); y2=atoi(argv[5]);
  }
if( argc > 1 )
        strcpy( datafile, argv[1] );
                           /* Read the data the from input */
init_screen( argc ) ;      /* file and dump it to thescreen*/
for(i=0;i<=55000;i++ );    /* A delay loop.                */
                           /* Store the original block size*/
cmprsfactor = ( float )bufr_size ;
init_table() ;             /* Initialize the tables.       */
                           /* Get the block from the screen*/
get( x1, y1, x2, y2, work_bufr ) ; /* memory then display  */
put( x1, y1, work_bufr ) ; /* it again. We have to move and*/
                           /* swap the bytes of the screen */
                           /* data  from work_bufr to      */
                           /* data_bufr since the latter is*/
                           /* the input to both the comp-  */
                           /* ression and the decompression*/
                           /* functions.                   */
swapbyts(FP_OFF(datafarptr), FP_SEG(datafarptr),
         FP_OFF(workfarptr), FP_SEG(workfarptr), bufr_size );
blksize=bufr_size;
setscmode(TEXT_MODE);
printf(" Compression is in progress \n" ) ;
tstart=gttime();           /* Record start of compression. */
                /* Count the run-lengths of black and white*/
                /* colors, where run-lengths are  limited  */
                /* between 1 and 128, in the screen  block */
                /* addressed by data_bufr+4. Put the code  */
                /* for each run-length in work_bufr. The   */
                /* size (in bytes) of the block is passed  */
                /* in bufr_size. The count of the symbols  */
                /* is returned by count_symbols() and      */
                /* stored in "temp".                       */
```

```
        temp=bufr_size =
                   count_symbols( work_bufr+4, data_bufr, bufr_size);
                        /* Compress the data in work_bufr+4 using  */
                        /* Lempel-Zev-Welch algorithm and return   */
                        /* the compressed data in the work_buffer. */
                        /* The  data_bufr  is  used  for  internal */
                        /* manipulation within compress() and other*/
                        /* functions  it  calls.  The  size of the */
                        /* compressed  buffer  is  returned  in    */
                        /* bufr_size.                              */
        compress( work_bufr+4, data_bufr, &bufr_size ) ;
        findtime( cmprstime )
        printf(" Now decompression is in progress \n" ) ;
        init_table() ;
        tstart=gttime();          /* Record start of decompression*/
                        /* Decompress the block compressed by LZW  */
                        /* algorithm.                              */
                        /* work_bufr = compressed buffer, as input,*/
                        /* and decompressed buffer, as output.     */
                        /* data_buffer = work  area  used  inside  */
                        /* decompress() and the function it calls. */
                        /* bufr_size = size of block compressed by */
                        /* LZW algorithm.                          */
        decompress((unsigned *)(work_bufr+4), data_bufr, bufr_size);
                        /* Find the run-lengths corresponding  to  */
                        /* the codes in the input work_bufr+4. Fill*/
                        /* data_bufr with the runs. temp = size of */
                        /* the symbols supplied by decompress() =  */
                        /* size of the result of count_symb().     */
        dcmprs_lzw( data_bufr, work_bufr+4, temp ) ;
        findtime( dcmprstime )
        movedata( FP_SEG(datafarptr), FP_OFF(datafarptr),
                  FP_SEG(workfarptr), FP_OFF(workfarptr), blksize);
        setscmode(HI_RES);        /* Display data on the screen to*/
        put( x1, y1, work_bufr );  /* make sure the program is    */
                                   /* working.                    */
        for( i=0; i<=55000; i++ ) ;
        setscmode(TEXT_MODE);
        passparmtrs( datafile, x1, y1, x2, y2, temp) ;
        print( cmprstime, dcmprstime,
                        cmprsfactor = cmprsfactor/bufr_size ) ;
}

/*------------------------- END main() ----------------------------*/
/*------------------------- END main.c ----------------------------*/
```

16.2. File Contsym.c

```
/*
 *===================================================================
 *
 * update_cmprsdblk(unsigned no of pels,int color)
 * screenbufr  = pointer to uncompressed block.
 * output      = pointer to output containing the symbols (byte each)
 *               for the encountered run-lengths of black and white
 *               pels.
 * currentword = pointer to current position, in words, in the
 *               uncompressed buffer.
 * nmbrwords   = word length of of the uncompressed buffer. We assume
 *               xsize is evenly divisible by 16, i.e.
 *               xsize (in pels) is an exact number of words.
 * color       = color of the pel.
 * pelcolor    = color of current pel (temporary storage).
 * word        = cuurent word in uncompressed line.
 * pelpos      = { 16 for leftmost pel} {1 for rightmost pel }.
 * blocksize   = size of uncounted (uncompressed) block, in bytes.
 *===================================================================
 */

#include        <stdio.h>
#include        <dos.h>
#define         LINT_ARGS
#define         BLACKBIT        0
#define         WHITEBIT        1
#define         ENDBITS         2
#define         update_cmprsdblk(pelcontr, color) \
                        { if(color==WHITEBIT) \
                             output[symbolcount++]= 127+pelcontr; \
                          else  output[symbolcount++]=pelcontr-1; \
                        }

void            swapbyts( unsigned *, unsigned *, unsigned ) ;

/* ==================== count_symbols() ======================== */

unsigned        count_symbols (output, screenbufr, bloksize)
char            *output, far *screenbufr;
unsigned        bloksize ;

{
unsigned        far  *currentword;
int             wordcount;
int             color,lastcolor;
unsigned        pelcontr=0, symbolcount;
register        unsigned        word,pelpos;
```

```
symbolcount = 0 ;
                                /* Assume blocksize= 16 * constant.*/
wordcount=bloksize/2 ;
currentword=(unsigned far *)screenbufr;
word=*currentword;
if ((word)&0x8000)              /* If the first pel is 1 then the  */
        { color=WHITEBIT; }     /* first color is white,           */
else                            /* else                            */
        {                       /* first  pel  is  zero  so  the   */
        color=BLACKBIT;         /* first color is black.           */
        word=~word;             /* Negate  the  word  so  our  way */
        }                       /* of  counting  will  work.       */
                                /* We count from left to right.    */
pelpos=16;
while(color<ENDBITS)            /* Do while not end of block.      */
        {                       /* Do while color is the same and  */
                                /* current word hasn't changed.    */
        while( (word&0x8000)&&(pelpos>0) )
                {
                pelcontr++;
                                /* If max run-length =128 of color */
                                /* then send its symbol to the     */
                                /* output.                         */
                if( pelcontr == 128 )
                        {
                        update_cmprsdblk(pelcontr,color)
                                /* Start counting again.           */
                        pelcontr = 0;
                        }
                pelpos--;       /* Decrease the count of unscanned */
                                /* pels in word. Move the next pel */
                word=word<<1;   /* to pel 16.                      */
                }
        if(pelpos>0)            /* If still inside the current word*/
                {
                                /* Make sure the last run-length   */
                                /* was not 128. Then output  the   */
                                /* symbol   of   the   current     */
                                /* run-length.                     */
                if( pelcontr > 0 )
                        update_cmprsdblk(pelcontr,color)
                word=~word;     /* Negate the word so we can check */
                                /* for the new color.              */
                                /* Flip the color to the new color.*/
                color=(color) ? 0 : 1;
                                /* Start counting the new pels.    */
                pelcontr=0;
                }
        else                    /* Else, all pels in current word  */
```

```
                {               /* were processed.                  */
                pelpos=16;      /* Start from the left most pel of */
                currentword++;  /* the next word.                   */
                                /* If  color is  black  we  need to*/
                                /* negate  word  so  our  way  of  */
                                /* counting  can  work.             */
                word= (color) ? *currentword : ~(*currentword);
                if(--wordcount==0)
                                /* If end of  block  then  output  */
                        {       /* the symbol of the run-length.   */
                                /* Make sure the last run-length   */
                                /* was not 128. Then output  the   */
                                /* symbol of the current           */
                                /* run-length.                     */
                        if( pelcontr > 0 )
                                {
                                update_cmprsdblk(pelcontr,color)
                                /* Signal  end  of  block  to  the */
                                /* outer  loop.                    */
                                color=ENDBITS;
                                }
                        }
                }
        }
                                /* Signal  the user if there was   */
                                /* an  error.                      */
if(color>ENDBITS)
        printf("****** error in color, color=%d /n",color);
                                /* Return  the number  of  symbols */
                                /* sent to the output = size of    */
                                /* "newblock".                     */
return( symbolcount ) ;
}
/* -------------------- END count_symbols() -------------------- */
/*----------------------- END contsym.c -----------------------*/
```

16.3. File Dcmpsym.c

```
/* ============================================================*/
/* Find the run-length for each symbol and send it to the output. */
/* size   = size (in bytes ) of input = number of symbols in input.*/
/* input  = pointer to buffer containing symbols of run-lengths.  */
/* output = pointer to buffer having data ready to be put on the  */
/*          screen.                                               */
/* dpelsremain = number of unfilled pels in current output byte.  */
/* currentbyteptr= pointer to current output byte.                */
/* ============================================================*/

#include        <memory.h>
```

```
#include        <dos.h>

static  int             dpelsremain;
static  unsigned char   far  *currentbyteptr;
static  unsigned char   rightpelsbyte[]={0,0x01,0x03,0x07,0x0f,
                                           0x1f,0x3f,0x7f,0xff};
static  unsigned char   leftpelsbyte[]={0,0x80,0xc0,0xe0,0xf0,
                                          0xf8,0xfc,0xfe,0xff};

mmset( unsigned, unsigned, char, unsigned );

/* ========================= dcmprs_lzw ========================== */
dcmprs_lzw( output, input, size )
unsigned        char    far  *output, *input ;
unsigned                size;
{

register        unsigned        input_index=0;
register        unsigned        code;

                                /* Set all output to black.        */
mmset( FP_OFF(output), FP_SEG(output), '\0', 16000 );
currentbyteptr=output;
dpelsremain=8;                  /* Fill bytes from left to right.  */
while( input_index < size )     /* Do while there is more code.    */
        {                       /* Get  the next  code.            */
        code = input[input_index++] ;
                                /* If it is a code for a white run-*/
                                /* length, output the run-length.  */
        if( code >= 128 )
                update_dcmprs_white( code-127 ) ;
                                /* Else, it is for a black run.    */
                                /* Output that run.                */
        else
                update_dcmprs_blak( code+1 ) ;
        }
}
/*------------------------ END lzw_dcmprs -------------------------*/


/* ===================== update_dcmprs_white ======================*/
/* It takes runs of white pels and output them to output, i.e.     */
/* fills the output with them.                                     */
/* ================================================================*/

update_dcmprs_white(clrpels)
                                /* Number of white  pels to store  */
                                /* in the output.                  */
register        int     clrpels;
{
```

```
register        int     difference;
                                /* Number  of bytes  we  can  fill */
unsigned        nmbrbytes;      /* with white completely.          */


difference=clrpels-dpelsremain;
                                /* If we can fill one or more byte */
                                /* completely  then, fill the pels */
                                /* remaining in the current byte.  */
if(clrpels >= (dpelsremain+8) )
        {
        *currentbyteptr |= rightpelsbyte[dpelsremain];
                                /* Find the number of bytes. We can*/
        nmbrbytes=(difference)>>3; /* fill them completely.        */
        ++currentbyteptr;
        mmset( FP_OFF(currentbyteptr),FP_SEG(currentbyteptr),
                                                0xff,nmbrbytes);
        currentbyteptr +=nmbrbytes.     /* Adjust the pointer.     */
                                /* If  difference  MOD 7  is  not  */
                                /* equal to zero  then  there  are */
                                /* still more pels that we did not */
                                /* outputed yet. So output them.   */
        if((difference=difference &0x7) !=0)
                *currentbyteptr=leftpelsbyte[(difference)];
                                /* In  the  new  byte  dpelsremain */
                                /*  = 8- pels outputed above.      */
        dpelsremain=8-(difference);
        }
                                /* Else, we can't  fill  any  byte */
                                /* completely.                     */
else
        {                       /* If dpelsremain > clrpels, it     */
                                /* means we can put the run inside */
        if(difference<0)        /* currentbyte.                    */
                {
                *(currentbyteptr) |= ( rightpelsbyte[clrpels] <<
                                           (dpelsremain-clrpels) );
                                /* Adjust dpelsremain accordingly. */
                dpelsremain -= clrpels;
                }
                                /* Else,   clrpels   have  to  be  */
                                /* outputted to more than one byte.*/
                                /* Fill the rest of the current    */
                                /* byte.                           */
        else
                {
                *currentbyteptr |=rightpelsbyte[dpelsremain];
                                /* Move to the next output byte and*/
                                /* and send to it the remaining of */
```

```
                                        /* clrpels.                       */
                    *(++currentbyteptr) =leftpelsbyte[difference];
                                        /* Account for last step.         */
                    dpelsremain=8- (difference);
                    }
          }
}
/*-------------------- END update_dcmprs_white --------------------*/


/* ===================== update_dcmprs_blak ======================*/
/* It take runs of black pels and output them to the output,i.e.   */
/* fill output with them.                                          */
/* It works exactly like update_dcmprs_white() except no filling   */
/* or outputing is done because output was initialized to zero at  */
/* start of dcmprs_lzw().                                          */
/* ===============================================================*/

update_dcmprs_blak(clrpels)
register        int     clrpels;
{

register        int     difference;
unsigned        nmbrbytes;

                                        /* Refer to comments above.       */
difference=clrpels-dpelsremain;
if(clrpels >= (dpelsremain+8) )
          {
          nmbrbytes=(difference)>>3;
          currentbyteptr +=nmbrbytes+1;
          dpelsremain=8-(difference &0x7 );
          }
else
          {
          if(difference<0)              /* Dpelsremain > clrpels.         */
                    dpelsremain -=clrpels;
          else
                    {
                    ++currentbyteptr;
                    dpelsremain=8- (difference);
                    }
          }
}
/*------------------- END update_dcmprs_black() -------------------*/
/*------------------------- END Dcmpsym.c -------------------------*/
```

16.4. File Mmset.asm

```
; A program to set the specified portion of memory to the given
; initial value. This is a replacement for the "memset" function
; provide by the run-time library of the MS C compiler. The main
; difference is that this function can be used to initialize a
; portion of memory out of the current segment i.e. pointed to by
; a far pointer.
; Inputs :
;        dest     : far pointer to destination.
;        chr      : character to set memory to.
;        bytecnt  : number of bytes .
NAME     MMSET
TITLE    MEMORY SET OF FAR DATA ITEMS
PUBLIC   _mmset

DEST_OFF  EQU   [BP+4]
DEST_SEG  EQU   [BP+6]
CHR       EQU   [BP+8]
BYTECONT  EQU   [BP+10]

DGROUP    GROUP CONST,  _BSS,   _DATA
          ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
_TEXT     SEGMENT BYTE PUBLIC 'CODE'
          ASSUME  CS:_TEXT

_mmset   PROC   NEAR
         PUSH   BP               ; SAVE THE REGISTERS
         MOV    BP,SP
         PUSH   DI
         PUSH   ES
         MOV    AX,DEST_SEG
         MOV    ES,AX
         MOV    DI,DEST_OFF
         MOV    BX,DI
         MOV    CX,BYTECONT
         JCXZ   DONE
         MOV    AL,CHR
         MOV    AH,AL
         MOV    DX,DI
         SHR    DX,1
         JNB    EVEN_OFFSET
         STOSB
         DEC    CX
EVEN_OFFS
         MOV    DX,CX
         SHR    CX,1
         REP    STOSW
         SHR    DX,1
         JNB    DONE
```

```
         MOV     BYTE PTR ES:[DI],AL
DONE:    MOV     AX,BX              ; RETURN THE POINTER TO THE
                                    ; DESTINATION.
         POP     ES                 ; RETRIEVE THE REGISTERS.
         POP     DI
         MOV     SP,BP
         POP     BP
         RET
_mmset   ENDP
_TEXT    ENDS
END
/*------------------------ END Mmset.asm --------------------------*/
```

16.5. File Swapfar.asm

```
NAME            SWAP
TITLE           SWAP BYTES IN EACH WORD IN SOURCE AND
;               PUT THE RESULT IN DESTINATION
PUBLIC          _swapbyts
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP

TO_OFFSET       EQU     [BP+4]
TO_SEGMENT      EQU     [BP+6]
FROM_OFFSET     EQU     [BP+8]
FROM_SEGMENT    EQU     [BP+10]
WORDCONT        EQU     [BP+12]

_TEXT      SEGMENT
_swapbyts       PROC    NEAR
                PUSH    BP
                MOV     BP,SP
                PUSH    DI
                PUSH    SI
                PUSH    ES
                PUSH    DS
                MOV     AX,FROM_SEGMENT
                MOV     DS,AX
                MOV     AX,TO_SEGMENT
                MOV     ES,AX
                MOV     CX,WORDCONT
                MOV     SI,FROM_OFFSET
                MOV     DI,TO_OFFSET
LOOP1:          LODSW
                XCHG    AH,AL
                STOSW
                LOOP    LOOP1
                POP     DS
                POP     ES
```

```
                POP     SI
                POP     DI
                MOV     SP,BP
                POP     BP
                RET
_swapbyts       ENDP
_TEXT   ENDS
END
/*------------------------ END Swapfar.asm -----------------------*/
```

17. APPENDIX F. PROGRAM LIST OF METHOD LZWB1

The files in this listing make use of the files in the following sections:

- Appendix B: 13.9, 13.11, and 13.12.
- Appendix D: 15.2 - 15.9.
- Appendix E: 16.1, 16.4, and 16.5.

17.1. File Dcmpsym.c

```
/*
 * Refer to the comments in file dcmpsym.c
 * in appendix E section 16.3.
 */

#include        <memory.h>
#include        <dos.h>
#define         uchar           unsigned char

static          int     dpelsremain;
static          uchar   far *currentbyteptr;
static          uchar   two_strings[]={
                                0x5,    0x9,    0xd,    0x11,
                                0x6,    0xa,    0xe,    0x12,
                                0x7,    0xb,    0xf,    0x13};
static          uchar   three_strings []= {
                                0x29,   0x2a,   0x2b,   0x2c,
                                0x49,   0x4a,   0x4b,   0x4c,
                                0x31,   0x51,   0x32,   0x52,
                                0x39,   0x59,   0x3a,   0x5a    };
uchar   *ptr_two_strings= two_strings;
uchar   *ptr_three_strings= three_strings;

mmset( unsigned, unsigned, char, unsigned );

/* ====================== dcmprs_lzw() ========================= */
dcmprs_lzw( output, input, size )
unsigned        char    far *output, *input ;
unsigned                size;

{
register        unsigned        input_index=0;
register        unsigned        code;
unsigned        char    temp;

mmset( FP_OFF(output), FP_SEG(output),'\0', 16000 );
currentbyteptr=output;
dpelsremain=8;
while( input_index < size )
        code = input[input_index++] ;
        if(code<200)
```

```
            {
            if( code >= 100 )
                update_dcmprs_white( code-99 ) ;
            else
                update_dcmprs_blak( code+1 ) ;
            }
        else
           {
           if(code < 224)
              {
              temp= two_strings[(code-200)%12];
              if (code <212)
                 {
                 update_dcmprs_blak(temp>>2);
                 update_dcmprs_white(temp&0x3);
                 }
              else
                 {
                 update_dcmprs_white(temp>>2);
                 update_dcmprs_blak(temp&0x3);
                 }
              }
           else
              {
              temp= three_strings[(code-224)%16];
              if(code<240)
                 {      /* bwb */
                 update_dcmprs_blak(temp>>5);
                 update_dcmprs_white((temp>>3)&0x3);
                 update_dcmprs_blak(temp&0x7);
                 }
              else
                 {
                 if(code<256)
                        { /* wbw */
                        update_dcmprs_white(temp>>5);
                        update_dcmprs_blak((temp>>3)&0x3);
                        update_dcmprs_white(temp&0x7);
                        }
                 else
                        { printf(" eror in dcmpsym code > 256");}
                 }
              }
           }
        }
}
/*--------------------- END lzw_dcmprs() ------------------------*/

static  unsigned char   rightpelsbyte[]={0,0x01,0x03,0x07,0x0f,
                                        0x1f,0x3f,0x7f,0xff};
```

```
static  unsigned char   leftpelsbyte[]={0,0x80,0xc0,0xe0,0xf0,
                                          0xf8,0xfc,0xfe,0xff};

/* =================== update_dcmprs_white() ====================*/
update_dcmprs_white(clrpels)
register        int     clrpels;
{

register        int     difference;
unsigned        nmbrbytes;

difference=clrpels-dpelsremain;
if(clrpels >= (dpelsremain+8) )
        {
        *currentbyteptr |= rightpelsbyte[dpelsremain];
        nmbrbytes=(difference)>>3;
        ++currentbyteptr;
        mmset( FP_OFF(currentbyteptr),FP_SEG(currentbyteptr),
                                             0xff,nmbrbytes);
        currentbyteptr +=nmbrbytes.
        if((difference=difference &0x7) !=0)
                *currentbyteptr=leftpelsbyte[(difference)];
        dpelsremain=8-(difference);
        }
else
        {
        if(difference<0)
                {
                *(currentbyteptr) |= ( rightpelsbyte[clrpels] <<
                                        (dpelsremain-clrpels) );
                dpelsremain -= clrpels;
                }
        else
                {
                *currentbyteptr |=rightpelsbyte[dpelsremain];
                *(++currentbyteptr) =leftpelsbyte[difference];
                dpelsremain=8- (difference);
                }
        }
}
/*----------------- END update_dcmprs_white() --------------------*/


/* =================== update_dcmprs_blak() =====================*/
/* It takes runs of black pels and output them to the output,i.e.  */
/* it fills the output with them.                                   */
/* It works exactly like update_dcmprs_white() except that no      */
/* filling or outputting is done because the output was initialized*/
/* to zero at the start of dcmprs_lzw().                            */
/* ===============================================================*/
```

```
update_dcmprs_blak(clrpels)
register        int     clrpels;
{

register        int     difference;
unsigned        nmbrbytes;

difference=clrpels-dpelsremain;
if(clrpels >= (dpelsremain+8) )
        {
        nmbrbytes=(difference)>>3;
        currentbyteptr +=nmbrbytes+1;
        dpelsremain=8-(difference &0x7 );
        }
else
        {
        if(difference<0)       /* dpelsremain > clrpels.           */
                dpelsremain -=clrpels;
        else
                {
                ++currentbyteptr;
                dpelsremain=8- (difference);
                }
        }
}
/*------------------- END update_dcmprs_black() ---------------------*/
/*------------------------ END  Dcmpsym.c --------------------------*/
```

17.2. File Contsym.c

```
/*
 * Refer to the comments in file contsym.c
 * in appendix E section 16.2.
 */

#include        <stdio.h>
#include        <dos.h>
#define         LINT_ARGS
#define         BLACKPEL        0
#define         WHITEPEL        1
#define         ENDPELS         2
#define         uchar           unsigned  char

void            init_cont_out(char *);
void            update_cmprsdblk(unsigned, int);
unsigned        int             find_code_2(uchar);
unsigned        int             find_code_3(uchar);
static          unsigned        symbolcount = 0 ;
```

```
/* ===================== count_symbols() ======================== */
unsigned        count_symbols (output, screenbufr, bloksize)
char            *output, far *screenbufr;
unsigned        bloksize ;

{
unsigned        far  *currentword;
int             wordcount;
int             color,lastcolor;
unsigned        pelcontr=0;
register        unsigned        word,pelpos;

init_cont_out(output);
wordcount=bloksize/2 ;
currentword=(unsigned far *)screenbufr;
word=*currentword;
if ((word)&0x8000)
        { color=WHITEPEL; }
else
        {
        color=BLACKPEL;
        word=~word;
        }
pelpos=16;
while(color<ENDPELS)
        {
        while( (word&0x8000)&&(pelpos>0) )
                {
                pelcontr++;
                if( pelcontr == 100 )
                        {
                        update_cmprsdblk(pelcontr,color)
                        pelcontr = 0;
                        }
                pelpos--;
                word=word<<1;
                }
        if(pelpos>0)
                {
                if( pelcontr > 0 )
                        update_cmprsdblk(pelcontr,color)
                word=~word;
                color=(color) ? 0 : 1;
                pelcontr=0;
                }
        else
                {
                pelpos=16;
```

```
                currentword++;
                word= (color) ? *currentword : ~(*currentword);
                if(--wordcount==0)
                        {
                        if( pelcontr > 0 )
                                {
                                update_cmprsdblk(pelcontr,color)
                                color=ENDPELS;
                                }
                        }
                }
        }
if(color>ENDPELS)
        printf("****** error in color, color=%d /n",color);
return( symbolcount ) ;
}
/* ------------------- END count_symbols() ---------------------- */

#define         two_strings_bw          ((unsigned ) (200-1))
#define         two_strings_wb          ((unsigned ) (212-1))
#define         three_strings_bwb       ((unsigned ) (224-1))
#define         three_strings_wbw       ((unsigned ) (240-1))
#define         start_two_strings(color)    \
                   ((color== 1) ? two_strings_bw : two_strings_wb)
#define         start_three_strings(color)   \
                   (color==0 ? three_strings_bwb : three_strings_wbw)

static          char            *cont_output;
static          int             string_num=1;
static          unsigned        s1,s2,s3;
static          uchar           temp;
static          unsigned        two_strings[]={
                                0x5,    0x9,    0xd,    0x11,
                                0x6,    0xa,    0xe,    0x12,
                                0x7,    0xb,    0xf,    0x13};
static          unsigned        three_strings []= {
                                0x29,   0x2a,   0x2b,   0x2c,
                                0x49,   0x4a,   0x4b,   0x4c,
                                0x31,   0x51,   0x32,   0x52,
                                0x39,   0x59,   0x3a,   0x5a    };

/*==================== update_cmprsdblk()========================*/
void    update_cmprsdblk(pelcontr, color)
unsigned        pelcontr;
int             color;

{
unsigned        code;

switch(string_num)
```

```
{
case 1  : {
          if(color)
                cont_output[symbolcount++]= 99 + pelcontr;
          else
                cont_output[symbolcount++]= pelcontr -1;
          if(pelcontr<=4)
                {
                string_num ++;
                s1=pelcontr;
                }
          break;
          }
case 2  : {
          if (pelcontr<=3)
                {
                temp=pelcontr | (s1<<2);
                cont_output[symbolcount -1]=
                  start_two_strings(color)+find_code_2(temp);
                if(s1<=2)
                    {
                    s2=pelcontr;
                    string_num++;
                    }
                else
                    string_num=1;
                }
          else
                {
                if(color)
                   cont_output[symbolcount++]= 99 + pelcontr;
                else
                   cont_output[symbolcount++]=  pelcontr -1;
                if(pelcontr<=4)
                    {
                    s1=pelcontr;
                    }
                else
                    string_num=1;
                }
          break;
          }
case 3  : {
          string_num=1;
          if( (s1+s2+pelcontr) <= 7 )
                {
                temp= pelcontr | (temp <<3);
                if (code=find_code_3(temp))
                   cont_output[symbolcount -1]=
                         code + start_three_strings(color);
```

```
                                else
                                   {
                                   if(color)
                                       cont_output[symbolcount++]=
                                                        99 + pelcontr;
                                   else
                                        cont_output[symbolcount++]=
                                                        pelcontr -1;
                                   if(pelcontr<=4)
                                        {
                                        string_num =2;
                                        sl=pelcontr;
                                        }
                                   }
                                }
                          else
                                {
                                if(color)
                                   cont_output[symbolcount++]= 99 + pelcontr;
                                else
                                   cont_output[symbolcount++]=  pelcontr -1;
                                if(pelcontr<=4)
                                        {
                                        string_num =2;
                                        sl=pelcontr;
                                        }
                                }
                          break;
                          }
                }
}
/* ------------------- END update_cmprsdblk()----------------------*/

/*===================== init_cont_out() ========================*/
void    init_cont_out(output)
char            *output;
{
cont_output=output;
}
/*--------------------- END init_cont_out() ----------------------*/
/*------------------------ END  Contsym.c ------------------------*/
```

17.3. File Scan2.asm

```
NAME            SCAN2
TITLE           SCANNING OF THE ALTERNATE TABLE TO FIND A MATCH
PUBLIC          _find_code_2

CHARCODE        EQU     [BP+4]     ; PASSED PARAMETER.
```

```
DGROUP          GROUP   CONST,  _BSS,   _DATA
                ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
_DATA           SEGMENT
EXTRN           _ptr_two_strings:WORD
_DATA           ENDS
_TEXT           SEGMENT BYTE PUBLIC 'CODE'

_find_code_2    PROC    NEAR
         PUSH   BP
         MOV    BP,SP
         PUSH   DI
         PUSH   ES
         MOV    AX,DS           ; INITIALIZE THE REGISTERS.
         MOV    ES,AX
         MOV    AX,CHARCODE
                                ; "CHARCODE" IS DEFINED IN
                                ; FILE DCMPSYM.C. DI = POINTER TO
                                ; THE TABLE HOLDING THE ELEMENTS
                                ; OF TYPE CHARACTER TO BE EXAMINED
                                ; IN THE SEARCH.
         MOV    DI,_ptr_two_strings
         MOV    CX,12           ; CX = SIZE OF TABLE.
                                ; SCAN THE CHAR_TABLE STARTING FROM
         REPNE  SCASB           ; DI UP TO CX ELEMENTS. STOP WHEN
                                ; THE FLAG BIT ZF IS SET TO 1
                                ; IF ZF= 0 WE FINISHED THE SCAN
         JNE    NOMATCH         ; BEFORE ANY MATCH. SO GO TO NOMATCH.
         MOV    AX,DI           ; ZF=1 SO WE HAD A MATCH. STORE
                                ; LENGTH OF SCANNED CHARACTERS IN AX.
         SUB    AX,_ptr_two_strings
         JMP    SCAN_DONE       ; SCAN IS DONE.
NOMATCH:
         XOR    AX,AX           ; NO MATCH SO RETURN VALUE=ZERO.
SCAN_DONE:
         POP    ES
         POP    DI
         MOV    SP,BP
         POP    BP
         RET
_find_code_2    ENDP
_TEXT           ENDS
END
/*----------------------- END  Scan2.asm ------------------------*/
```

17.4. File Scan3.asm

```
; REFER TO COMMENTS IN FILE SCAN2.C OF THIS APPENDIX.
NAME            SCAN3
```

```
TITLE           SCANNING OF THE THREE STRING TABLES TO FIND A MATCH.
PUBLIC          _find_code_3

CHARCODE        EQU     [BP+4]          ; PASSED PARAMETER.

DGROUP          GROUP   CONST,  _BSS,   _DATA
                ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP
_DATA           SEGMENT
EXTRN           _ptr_three_strings:WORD
_DATA           ENDS

_TEXT           SEGMENT BYTE PUBLIC 'CODE'
_find_code_3    PROC    NEAR
         PUSH   BP
         MOV    BP,SP
         PUSH   DI
         PUSH   ES
         MOV    AX,DS
         MOV    ES,AX
         MOV    AX,CHARCODE

         MOV    DI,_ptr_three_strings
         MOV    CX,16
         REPNE  SCASB
         JNE    NOMATCH
         MOV    AX,DI
         SUB    AX,_ptr_three_strings
         JMP    SCAN_DONE
NOMATCH: XOR    AX,AX
SCAN_DONE:
         POP    ES
         POP    DI
         MOV    SP,BP
         POP    BP
         RET
_find_code_3    ENDP
_TEXT           ENDS
END
/*------------------------ END  Scan3.asm -------------------------*/
```

18. APPENDIX G. PROGRAM LIST OF METHOD LZWB2

The files in this listing make use of the files in the following sections:

- Appendix B: 13.9, 13.11, and 13.12.
- Appendix D: 15.2 - 15.9.
- Appendix E: 16.1 - 16.5.

18.1. File Dcmprs.c

```
/*
 * Refer to the comments in file dcmprs.c
 * in appendix D section 15.3.
 */
#include        <memory.h>
#include        <stdio.h>
#include        <malloc.h>
#define         uchar           unsigned char
#define         MAX_SIZE        4096
#define         ST_MAX          1000
#define         SCRN_SIZE       16004
#define         ALPHABET_SIZE   256
#define         update_string_table()   \
                        {if( next_code < MAX_SIZE )\
                                {char_table[next_code]=code.k ;\
                                int_table[next_code]  = oldcode;\
                                next_code++ ;}}
#define         look_up()  \
                        {code.w = int_table[CODE] ;\
                         code.k = char_table[CODE];}

extern  unsigned        int_table[] ;
extern  uchar           char_table[] ;
extern  int             next_code;
static  char            *stack ;
static  unsigned        stack_index=0;

char    pop();
void    readjust_input(uchar far *, uchar *, unsigned, unsigned *);

/*========================== decompress() ==========================*/
decompress(decmprs_io, decmprs_work ,inputsize)
char            *decmprs_io,far  *decmprs_work;
unsigned        inputsize;

{
        unsigned        input_index=0;
        unsigned        oldcode,incode;
        unsigned        newsize ;
        char            temp;
        register        unsigned        output_index=0;
```

```
register        unsigned        CODE ;
char            finchar ;
struct  {
        char            k;
        unsigned        w;
        } code;
char            *temp_ptr;
unsigned  far   *input;
char            *databufr;

stack = malloc( ST_MAX );
readjust_input(decmprs_work,decmprs_io,inputsize,&newsize);
inputsize=(newsize/2);
input= (unsigned far *) decmprs_work;
databufr=decmprs_io;
CODE= oldcode= input[input_index++];
look_up()            /* -- MACRO -- find "code" components.*/
if(CODE >= ALPHABET_SIZE)       /* First code = "w,k".     */
        {
        if(code.w < ALPHABET_SIZE)
                databufr[output_index++]=code.w;
        else
                {
                temp=code.k;
                CODE=code.w;
                look_up()
                databufr[output_index++]=code.w;
                databufr[output_index++]=code.k;
                code.k=temp;
                }
        }
databufr[output_index]=finchar=code.k;
while(input_index< inputsize  )
{
        CODE=incode=input[input_index++];
        if(CODE >= next_code)
        {
                push(finchar);
                CODE=oldcode;
        }
        look_up()
        while(code.w!=0xffff)
        {
                push(code.k);
                CODE=code.w;
                look_up()
        }
        databufr[++output_index]=code.k;
        finchar=code.k;
        while(stack_index)
```

```
                    {
                    databufr[++output_index]=pop();
                    }
                update_string_table()
                oldcode=incode;
        }
}
/*------------------------ END decompress() -----------------------*/

/*============================ push() ===========================*/
push( item )
char    item;

{
        if( stack_index >= ST_MAX )
        {
                printf( " stack overflow in push \n" ) ;
                return ;
        }
        stack[stack_index++] = item ;

}
/*-------------------------- END push() -------------------------*/

/*============================ pop() ============================*/
char    pop()
{
        if( --stack_index < 0 )
        {
                printf(" Stack underflow in pop \n" ) ;
                return ('\0');
        }
        return stack[stack_index];
}
/*-------------------------- END pop() --------------------------*/
/*------------------------ END Dcmprs.c -------------------------*/
```

18.2. File Tables.c

```
/*
 * Refer to the comments in file tables.c
 * in appendix D section 15.4.
 */
#include        <stdio.h>
#include        <memory.h>
#include        <malloc.h>
#define         MAX_SIZE        4096
#define         ALPHABET_SIZE   256
#define         uchar           unsigned char
```

```
                                /* Define global varriables.        */
unsigned                int_table[MAX_SIZE] ;
unsigned        char    char_table[MAX_SIZE] ;
int                     next_code ;
unsigned                *ptr_int_table=int_table;
unsigned        char    *ptr_char_table=char_table;
unsigned                extracalls=0 ;

/*========================== init_table() =========================*/
init_table()
{
        register        int     index ;
        char            *datafile= "etables.dat";
        char            c;
        FILE            *in;
        unsigned        temp,*ptr_temp=&temp;

        memset( (char *) int_table,0xff,MAX_SIZE*2);
        for( index=0; index < ALPHABET_SIZE; index++ )
                char_table[index] = ( short ) index ;

                                /* Open  file to read data from.    */
        if( (in=fopen(datafile,"r")) != NULL )
        {
                for( index=256; index<312; index++)
                        {
                        fscanf(in,"%u%u",&int_table[index],ptr_temp);
                        char_table[index]=temp;
                        }
                if( ferror(in) )
                        {       /* If  any  error  was encountered */
                                /* while  reading  the  data then  */
                                /* inform us and exit.             */
                        printf(" Error in reading tables \n");
                        exit(0);
                        }
                fclose(in);
        }
        else                    /* File couldn't  be  opened  for  */
                                /* some reasons.                   */
                {
                printf(" ERROR ----- Can't open input file");
                exit(0);
                }
        next_code = index;
}
/*----------------------- END init_table() -------------------------*/
/*------------------------- END Tables.c ---------------------------*/
```

19. APPENDIX H. PROGRAM LIST OF METHOD LZW1

The files in this listing make use of the files in the following sections:

- Appendix B: 13.9, 13.11, and 13.12.
- Appendix D: 15.1 - 15.9.

19.1. File Tables.c

```
#include        <stdio.h>
#include        <memory.h>
#include        <malloc.h>
#define         MAX_SIZE        4096
#define         ALPHABET_SIZE   256
#define         uchar           unsigned char

                                /* Definition of GLOBAL          */
                                /* variables.                    */
unsigned        w1_table[MAX_SIZE] ;
unsigned        w2_table[MAX_SIZE] ;
uchar           w3_table[MAX_SIZE] ;
unsigned        *ptr_w1_table=w1_table;
unsigned        *ptr_w2_table=w2_table;
uchar           *ptr_w3_table=w3_table;
int             next_code ;
unsigned        extracalls=0 ;

/*========================== init_table() ==========================*/
/* This function initializes every element in int_table to a com-  */
/* bination that will  never occur. Since the code is only 12 bits */
/* long then the 16 bits used to hold these codes are to be <=     */
/* 0xfff. For this reason in this program the 0xffff code is used  */
/* to solve the above problem. It should be noted that any combin- */
/* ation > 0xfff should work correctly as well.                    */
/* Then the first 256 symbols in w2_table are initialized to 0-255.*/
/*=================================================================*/
init_table()

{
        register        int     index ;
                                /* Set  every  byte  in  the     */
                                /* int_table  to  0xffff  (i.e.  */
                                /* every code word = 0xffff) so  */
                                /* that no code will match with  */
                                /* it, because the actual codes  */
                                /* are only 12 bits.             */
        memset( (char *) w1_table,0xff,MAX_SIZE*2);
                                /* Set 1st 256 of char_table to  */
                                /* be the extended ASCII codes.  */
        for( index=0; index < ALPHABET_SIZE; index++ )
                w2_table[index] =  index ;
```

```
        next_code = ALPHABET_SIZE;
}
/*------------------------- END init_table() ------------------------*/
/*--------------------------- END Tables.c -------------------------*/
```

19.2. File Cmprs.c

```
#include        <memory.h>
#include        <malloc.h>
#define         uchar           unsigned        char
#define         MAX_SIZE        4096
#define         SCRN_SIZE       16004
#define         update_tables(a,b,c)    { wl_table[next_code] = a;\
                                          w2_table[next_code] = b;\
                                          w3_table[next_code] = c;\
                                          next_code++ ;}
#define         look_table2(w2,codec)   { w2=w2_table[codec]; }

extern  unsigned  wl_table[] ;  /* wl_table[],   w2_table[],       */
extern  unsigned  w2_table[] ;  /* w3_table[]  and  next_code are  */
extern  uchar     w3_table[] ;  /* defined  in tables.c.           */
extern  int       next_code  ;
extern  unsigned  extracalls ;
extern  unsigned  stack[];
extern  int       st_index ;    /* Stack size.                     */

void    adjust_output( uchar *, uchar far *, unsigned, unsigned * );
void    decompose(unsigned );

/*========================= compress() ==================================*/
compress( compress_io,compress_work, ptr_bufr_size )
uchar           *compress_io, far *compress_work ;
unsigned        *ptr_bufr_size ;

{
        uchar     *input;
        unsigned  far *output;
        char      *ptr_new_output;
        unsigned  bufr_size;
        unsigned  code ;
        register  unsigned      data_index=0  ;
        unsigned  out_index=0 ;
        struct    {
                  unsigned      wl;
                  unsigned      w2;
                  }       string ;
        uchar     w3, first_ch;
        unsigned  Li, Lj ;
        unsigned  position, indexl ;
```

```
register  unsigned      j ;

input=compress_io;
output=(unsigned far *)compress_work;
                        /* Li = first input element.       */
Li= input[data_index++] ;
                        /* Lj = second input element.      */
Lj=  input[data_index++] ;
first_ch = Lj ;
output[out_index++] = Li;
string.w1 = Lj;
w3 = Lj ;
                        /* Find bufr_size.                 */
bufr_size=*ptr_bufr_size;
                        /* Loop while there is more input. */
while( data_index <  bufr_size )
  {                     /* Search for the largest block in */
                        /* w1_table.                       */
   while( data_index <  bufr_size )
        {               /* Get 2nd element in the new block*/
        string.w2=input[data_index++] ;
                        /* See if w1.w2 is in tables.      */
         if( scan_w2( string.w1, string.w2, &code) )
                        /* w1.w2 is in the tables, so let  */
                        /* new w1 = w1.w2.                 */
            string.w1=code;
         else           /* w1.w2 was not in the tables.    */
          {             /* First element of 2nd block = w2.*/
            first_ch=string.w2 ;
                        /* Go to the second while loop and */
                        /* search for a table entry that   */
                        /* has w1 and its w2 starts with w3*/
            break ;
          }
        }
                        /* We  already  searched  for  two */
                        /* elements or more, so  start     */
   position = 256 ;     /* searching after 256.            */
   while( data_index < bufr_size )
        {
         if( scan_w3(string.w1, first_ch, &code, position) )
          {             /* Start searching after code.     */
            position=code+1;
            look_table2(string.w2, code )
                        /* st_index points to the last     */
                        /* element on the stack.           */
            decompose( string.w2 ) ;
            index1 = data_index ;
            if( (bufr_size - index1) >= st_index )
               {        /* data_index is already pointing  */
```

```
                        /* to the element after w3 in the  */
                        /* input so there is no need to    */
                        /* compare it. The "for" loop will */
                        /* start comparing  from  index1   */
                        /* that should be equal to stack[1]*/
                for(j=1;(j <= st_index) &
                              (input[index1++]==stack[j]) ; )
                   {
                    j++;
                   }
                if( j == (st_index+1) )
                  {
                   string.w1=code ;
                        /* data_index === w3+1.            */
                   data_index += st_index  ;
                   first_ch=input[data_index++] ;
                  }
                else
                  {   ;
                  }
               }
           }
         else
            break ;
        }
   Lj = string.w1;
   output[out_index++] = Lj;
                        /* If the tables are not full yet, */
   if(next_code<MAX_SIZE)
                        /* then string --> string table,   */
                        /* i.e put w and k in the w1_table */
                        /* and w2_table respectively at the*/
                        /* position indexed by next_code.  */
        update_tables( Li, Lj, w3 )
   else
        extracalls++ ;
   Li = Lj ;
   string.w1 = first_ch ;
   w3 = first_ch ;
  }
                        /* Make sure the last symbol was   */
                        /* sent to the output.             */
if( data_index == bufr_size )
        {
        output[out_index] = input[bufr_size-1] ;
        out_index++;
        }
                           /* Pack the output codes from a */
                           /* string of words format to a  */
                           /* string  of  12  bits  codes  */
```

```
                                   /* format. The input to      */
                                   /* adjust_output() is compress_ */
                                   /* work. It sends the output in */
                                   /* the final form in compress_io*/
        adjust_output(compress_io ,compress_work,
                                     2*out_index , ptr_bufr_size ) ;
}
/*----------------------- END compress() ------------------------*/
/*------------------------- END Cmprs.c --------------------------*/
```

19.3. File Dcmprs.c

```
#include        <memory.h>
#include        <stdio.h>
#include        <malloc.h>
#define         uchar           unsigned char
#define         MAX_SIZE        4096
#define         ST_MAX          1000
#define         SCRN_SIZE       16004
#define         update_w12_table(w1,w2) \
                        { w1_table[next_code] = w1;\
                          w2_table[next_code] = w2;\
                          next_code++ ;}

                                   /* W1_table, w2_table and next_code*/
extern  unsigned  w1_table[] ;     /* are defined globally in       */
extern  unsigned  w2_table[] ;     /* tables.c.                     */
                                   /* Index of the next code in tables*/
extern  int       next_code;       /* not used yet.                 */
extern  unsigned  stack[];
                                   /* First unused element. Stack   */
extern  unsigned  st_index;        /* grows upward.                 */

void    readjust_input(char far *, char *, unsigned, unsigned * ) ;

/*====================== decompress() =========================*/
/* Input is in the form of 12 bits codes stored serialy. We have  */
/* to readjust them to integer format so we can store them and use */
/* them in the w1_table.                                           */
/* Inputsize is size of input in bytes.                            */
/* decmprs_io= as input to decmprs it points to compressed data.   */
/* decmprs_io= as ouput of decmprs it points to decompressed data. */
/* decmprs_work= pointer to a temporary area.                      */
/*==============================================================*/
decompress(decmprs_io, decmprs_work ,inputsize)
char            *decmprs_io, far *decmprs_work;
unsigned        inputsize;

{
```

```
        unsigned  input_index=0;
                                   /* Size  of the  compressed  data   */
                                   /* stored in a word form for each   */
                                   /* code. It is equal to the size    */
        unsigned  newsize ;        /* of readjust_input() output.      */
        register  unsigned         output_index=0;
        register  unsigned         j ;
        unsigned  w1, w2;
        unsigned  far *input;
        char      *databufr;

                                   /* Adjust the input from 12 bits    */
                                   /* serial codes into an array of    */
                                   /* integers and then put the  size  */
                                   /* of the  array  in newsize.       */
        readjust_input(decmprs_work,decmprs_io,inputsize,&newsize);
        inputsize=(newsize/2);  /* Find size of input code in words*/
        input= (unsigned far *) decmprs_work;
        databufr=decmprs_io;
        w1 =  input[input_index++] ;
        databufr[output_index++] = w1 ;
        while( input_index < inputsize )
                {
                w2=input[input_index++] ;
                decompose( w2 ) ;
                j=0 ;
                do
                        {
                        databufr[output_index++] = stack[j++] ;
                        }
                while( j <= st_index )    ;
                if( next_code < MAX_SIZE )
                        update_w12_table(w1,w2);
                w1 = w2 ;
                }
        printf("\n");
}
/*----------------------- END decompress() ----------------------*/
/*------------------------- END Dcmprs.c ------------------------*/
```

19.4. File Dcompose.c

```
#define         TRUE    1
#define         FALSE   0
#define         MAX_SIZE        4096
#define         look_up_w12(xw1,xw2,codec)  \
                     { xw1=w1_table[codec]; \
                       xw2=w2_table[codec]; }
```

```
extern          unsigned        w1_table[] ;
extern          unsigned        w2_table[] ;
unsigned        stack[MAX_SIZE];
int             st_index ;      /* Stack size.                       */

/*======================= decompose() ============================*/
void    decompose( code )
unsigned        code ;

{
int             strngstk = 0 ;
register        unsigned        w1, w2;
unsigned        loop1,loop2 , strng[500];

        if(code<256)
                {
                stack[st_index=0]=code;
                return;
                }
        st_index     = 0 ;
        do
            {
            loop1=TRUE;
            while( loop1)
                {
                 look_up_w12( w1, w2, code )
                 strng[strngstk++] =  w2 ;
                 if(  w1 < 256 )
                    {
                     stack[st_index++] =  w1 ;
                     loop1= FALSE;
                    }
                 else
                     code =  w1 ;
                }
             loop2=TRUE;
             while( (loop2) & (strngstk>0) )
                {
                 w2 = strng[--strngstk] ;
                 if(  w2 < 256 )
                     stack[st_index++] =  w2 ;
                 else
                    {
                     code =  w2 ;
                     loop2=FALSE;
                    }
                }
            }
            while( strngstk > 0 | (!loop2));
        st_index-- ;
```

```
}
/*------------------------ END decompose() ------------------------*/
/*------------------------ END Dcompose.c -------------------------*/
```

19.5. File Scanw2.asm

```
; INPUT : ( PARAMETERS PASSED BY CALLING SUBROUTINE )
;       1) W2_CODE  = CHARACTER PART OF THE CODE,i.e K.
;       2) W1_CODE  = UNSIGNED INTEGER PART OF THE CODE, i.e. W.
;       3) CODEADRS = ADDRESS OF CODE ,i.e. WHERE WE RETURN THE
;                     CODE WHICH HAS W AND K EQUAL TO "INTCODE" AND
;                     "CHARCODE" RESPECTIVELY.
;
; OUTPUT :
;       1) THE FUNCTION RETURN VALUE = 1 IF A MATCH IS FOUND.
;                                      0 IF NO MATCH.
;
; THE FUNCTION NEEDS TO SHARE THE FOLLOWING VARIABLES WITH WHOEVER
; HAS THEM:
;       1) _ptr_w2_table = A POINTER TO FIRST ELEMENT IN CHAR_TABLE.
;       2) _ptr_w1_table  = A POINTER TO FIRST ELEMENT IN INT_TABLE.
;       3) next_code  = NUMBER OF FIRST FREE CODE IN CHAR_TABLE.
;                     = NUMBER OF FIRST FREE CODE IN INT_TABLE.
NAME     SCAN
TITLE    SCANNING OF THE W1 AND W2 TABLES TO FIND A MATCH
PUBLIC   _scan_w2

w1       EQU    [BP+4]          ; PASSED PARAMETERS.
w2       EQU    [BP+6]
ptr_code EQU    [BP+8]

DGROUP   GROUP  CONST,  _BSS,   _DATA
         ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP

_DATA    SEGMENT
EXTRN    _ptr_w2_table:WORD
EXTRN    _ptr_w1_table:WORD
EXTRN    _next_code:WORD
_DATA    ENDS

_scan_w2 PROC   NEAR
         PUSH   BP
         MOV    BP,SP
         PUSH   DI
         PUSH   SI
         PUSH   ES
         MOV    AX,DS
         MOV    ES,AX
                                ; INITIALIZE THE REGISTERS TO THE
```

```
        MOV    AX,w1             ; CORRESPONDING PARAMETERS PASSED
        MOV    DX,w2             ; FROM THE CALLING PROGRAM.
                                 ; SI = POINTER TO THE TABLE HOLDING
                                 ; ELEMENTS OF CHARACTER TYPE. THIS
                                 ; TABLE HOLDS THE SECOND PART TO BE
                                 ; EXAMINED IN THE SEARCH.
        MOV    SI,_ptr_w2_table

                                 ; DI = POINTER TO THE TABLE USED IN
                                 ; THE SEARCH. IT HOLDS THE INTEGER
                                 ; PART WE SCAN FOR.
        MOV    DI,_ptr_w1_table
        MOV    CX,_next_code     ; CX = NEXT NUMBER NOT USED IN THE
                                 ; TABLES YET.
LOOP1:                           ; SACN THE WORD TABLE STARTING
        REPNE  SCASW             ; FROM DI UP TO CX ELEMENTS.
                                 ; IF ZF= 0 WE FINISHED THE SCAN
        JNE    NOMATCH           ; BEFORE ANY MATCH. SO GO TO NOMATCH.
        MOV    BX,DI             ; ZF=1 SO WE HAD A MATCH. STORE THE
                                 ; LENGTH OF THE SCANNED WORDS IN BX.
        SUB    BX,_ptr_w1_table
                                 ; GET NUMBER OF SCANNED WORDS.
        SUB    BX,2              ; ADJUST FOR LOOP INDEX STEPPING
                                 ; ONE MORE WORD.
                                 ; SINCE WE HAD A WORD MATCH,
        CMP    DX,[BX+SI]        ; SEE IF WE HAVE CHAR MATCH.
                                 ; IF YES THEN WE HAVE A COMPLETE
        JE     MATCH             ; MATCH. SO GO TO MATCH.
                                 ; CHAR DID NOT MATCH SO TRY AGAIN
                                 ; AS LONG AS CX (= REMAINING CODES TO
                                 ; BE SEARCHED ) NOT EQUAL TO ZERO.
                                 ; IF CX REACHED ZERO BEFORE WE HAD
                                 ; ANY MATCH THEN "JNE  NOMATCH" WILL
        JMP    LOOP1             ; DROP US TO NOMATCH:
NOMATCH:
        MOV    AX,0              ; NO MATCH SO RETURN ZERO IN AX.
        JMP    SCAN_DONE         ; SCAN IS DONE.
MATCH:                           ; THERE WAS A MATCH SO MAKE DI =
                                 ; LENGTH OF SCANNED WORDS.
        SUB    DI,_ptr_w1_table
        SHR    DI,1              ; MAKE DI = NUMBER OF SCANNED WORDS.
        DEC    DI                ; ADJUST FOR LOOP INDEX STEPPING
                                 ; ONE MORE WORD.
        MOV    BX,ptr_code
        MOV    [BX],DI
        MOV    AX,1
SCAN_DONE:
        POP    ES
        POP    SI
        POP    DI
```

```
         MOV     SP,BP
         POP     BP
         RET
_scan_w2         ENDP
_TEXT            ENDS
END
/*----------------------- END Scanw2.asm ------------------------*/
```

19.6. File Scanw3.asm

```
;  REFER TO COMMENTS IN FILE SCANW2.ASM IN THIS APPENDIX.
NAME             SCAN_W3
TITLE            SCANNING OF THE
PUBLIC           _scan_w3

w1       EQU     [BP+4]            ; PASSED PARAMETERS.
w3       EQU     [BP+6]
ptr_code EQU     [BP+8]
position EQU     [BP+10]

DGROUP   GROUP  CONST,  _BSS,   _DATA
         ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP

_DATA    SEGMENT
EXTRN    _ptr_w3_table:WORD
EXTRN    _ptr_w1_table:WORD
EXTRN    _next_code:WORD
_DATA    ENDS

_scan_w3         PROC    NEAR
         PUSH    BP
         MOV     BP,SP
         PUSH    DI
         PUSH    SI
         PUSH    ES
         MOV     AX,DS
         MOV     ES,AX
         MOV     AX,w1
         MOV     DL,w3
         MOV     SI,_ptr_w3_table
         MOV     BX,position
         MOV     DI,_ptr_w1_table
         SHL     BX,1
         ADD     DI,BX
         MOV     CX,_next_code
         SUB     CX,position
         JZ      NOMATCH
LOOP1:
         REPNE   SCASW
```

```
         JNE     NOMATCH
         MOV     BX,DI
         SUB     BX,_ptr_wl_table
         SHR     BX,1
         DEC     BX
         CMP     DL,[BX+SI]
         JE      MATCH
         JMP     LOOP1
NOMATCH:
         MOV     AX,0
         JMP     SCAN_DONE
MATCH:
         SUB     DI,_ptr_wl_table
         SHR     DI,1
         DEC     DI
         MOV     BX,ptr_code
         MOV     [BX],DI
         MOV     AX,1
SCAN_DONE:
         POP     ES
         POP     SI
         POP     DI
         MOV     SP,BP
         POP     BP
         RET
_scan_w3         ENDP
_TEXT            ENDS
END
/*------------------------ END Scanw3.asm -------------------------*/
```

20. APPENDIX I. PROGRAM LIST OF METHOD LZW2

The files in this listing make use of the files in the following sections:

- Appendix B: 13.9, 13.11, and 13.12.
- Appendix D: 15.1 - 15.9.
- Appendix H: 19.1 and 19.3 - 15.6.

20.1. File Cmprs.c

```
#include        <memory.h>
#include        <malloc.h>
#define         uchar           unsigned        char
#define         MAX_SIZE        4096
#define         SCRN_SIZE       16004
#define         TRUE            1
#define         FALSE           0
#define         update_tables(a,b,c)    { w1_table[next_code] = a;\
                                          w2_table[next_code] = b;\
                                          w3_table[next_code] = c;\
                                          next_code++ ;}
#define         look_table2(w2,codec)   { w2=w2_table[codec]; }

extern  unsigned  w1_table[] ;  /* w1_table[],   w2_table[],      */
extern  unsigned  w2_table[] ;  /* w3_table[]  and  next_code are */
extern  uchar     w3_table[] ;  /* defined  in tables.c.          */
extern  int       next_code  ;
extern  unsigned  extracalls ;
extern  unsigned  stack[MAX_SIZE];
extern  int       st_index ;    /* Stack size.                    */

void    adjust_output( uchar *, uchar far *, unsigned, unsigned * );
void    decompose(unsigned );

/*===================== compress() ============================*/
compress( compress_io,compress_work, ptr_bufr_size )
uchar           *compress_io, far *compress_work ;
unsigned        *ptr_bufr_size ;

{
        uchar     *input;
        unsigned  far *output;
        char      *ptr_new_output;
        unsigned  bufr_size;
        unsigned  code ;
        register  unsigned      data_index=0  ;
        unsigned  out_index=0 ;
        struct    {
                  unsigned      w1;
                  unsigned      w2;
                  }      string ;
```

```
uchar     w3, first_ch;
unsigned  Li, Lj ;
unsigned  longblk, loop3 ;
int       bigstk ;
unsigned  position, indexl ;
register  unsigned      j ;

input=compress_io;
output=(unsigned far *)compress_work;
                       /* Li = first input element        */
Li= input[data_index++] ;
                       /* Lj = second input element.      */
Lj=  input[data_index++] ;
first_ch = Lj ;
output[out_index++] = Li;
string.wl = Lj;
w3 = Lj ;
                       /* Find bufr_size.                 */
bufr_size=*ptr_bufr_size;
                       /* Loop while there is more input. */
while( data_index <  bufr_size )
  {                    /* Search for the largest block in */
                       /* wl_table.                       */
   while( data_index <  bufr_size )
        {              /* Get 2nd element in the new block*/
        string.w2=input[data_index++] ;
                       /* See if wl.w2 is in tables.      */
         if( scan_w2( string.wl, string.w2, &code) )
                       /* wl.w2 is in the tables, so let  */
                       /* new wl = wl.w2.                 */
            string.wl=code;
         else          /* wl.w2 was not in the tables.    */
           {           /* First element of 2nd block = w2.*/
            first_ch=string.w2 ;
                       /* Go to the second while loop and */
                       /* search for a table entry that   */
                       /* has wl and its w2 starts with w3*/
            break ;
           }
        }
                       /* We  already  searched  for  two */
                       /* elements or more, so  start     */
   position = 256 ;    /* searching after 256.            */
   while( data_index < bufr_size )
        {
         if( scan_w3(string.wl, first_ch, &code, position) )
           {           /* Start searching after code.     */
            longblk = string.wl ;
            bigstk  = -1 ;
            loop3 = TRUE ;
```

```
            while( loop3 )
                {
                look_table2(string.w2, code )
                        /* st_index points to last element */
                        /* in stack.                       */
                decompose( string.w2 ) ;
                index1 = data_index ;
                if( ((bufr_size - index1) >= st_index ) &
                                ( st_index > bigstk ) )
                    {   /* Data_index is already pointing  */
                        /* to the element after w3 in the  */
                        /* input so no need to compare it. */
                        /* The  for  loop  will  start     */
                        /* comparing  from  index1  which  */
                        /* should be equal to stack[1].    */
                    for(j=1;(j <= st_index) &
                                (input[index1++]==stack[j]);)
                        {
                        j++;
                        }
                    if( j == (st_index+1) )
                        {
                        bigstk = st_index ;
                        longblk = code ;
                        }
                    }
                position = code + 1;
                if( scan_w3(string.w1, first_ch,
                                        &code, position) )
                        ;
                else
                    loop3 = FALSE ;
                }
             if( string.w1 == longblk )
                 break ;
             else
                {
                string.w1 = longblk ;
                position = longblk + 1 ;
                data_index += bigstk ;
                first_ch = input[data_index++] ;
                }
            }
        else
            break ;
        }
Lj = string.w1;
output[out_index++] = Lj;
                        /* If the tables are not full yet, */
if(next_code<MAX_SIZE)
```

```
                                /* then string --> string table,   */
                                /* i.e put w and k in the w1_table */
                                /* and w2_table respectively at the*/
                                /* position indexed by next_code.  */
                update_tables( Li, Lj, w3 )
           else
                extracalls++ ;
           Li = Lj ;
           string.w1 = first_ch ;
           w3 = first_ch ;
          }
                                /* Make sure the last symbol was   */
                                /* sent to the output.             */
        decompose( output[out_index - 1] );
        if( data_index == bufr_size )
                {
                output[out_index] = input[bufr_size-1] ;
                out_index++;
                }
                                   /* Pack the output codes from a */
                                   /* string of words format to a  */
                                   /* string  of  12  bits  codes  */
                                   /* format.  The  input  to      */
                                   /* adjust_output() is compress_ */
                                   /* work. It sends the output in */
                                   /* the final form in compress_io*/
        adjust_output(compress_io ,compress_work,
                                      2*out_index , ptr_bufr_size ) ;
}
/*------------------------ END compress() -------------------------*/
/*-------------------------- END Cmprs.c --------------------------*/
```

21. APPENDIX J. PROGRAM LIST OF METHOD LZW3

The files in this listing make use of the files in the following sections:

- Appendix B: 13.9, 13.11, and 13.12.
- Appendix D: 15.1 - 15.9.
- Appendix H: 19.3 and 15.4.

21.1. File Cmprs.c

```
#include        <memory.h>
#include        <malloc.h>
#define         uchar           unsigned        char
#define         MAX_SIZE        4096
#define         SCRN_SIZE       16004
#define         TRUE            1
#define         FALSE           0
#define  update_tables(a,b,c)
                { w1_table[next_code] = a;\
                  w2_table[next_code]  = b;\
                  ((char *) w4_table)[(2*next_code)+1] = c.second;\
                  ((char *) w4_table)[2*next_code]     = c.first;\
                  next_code++ ;}

extern   unsigned  w1_table[] ; /* w1_table[],   w2_table[],       */
extern   unsigned  w2_table[    /* w4_table[]  and  next_code are  */
extern   uchar     w4_table[]   /* defined  in tables.c.           */
extern   int       next_code  ;
extern   unsigned  extracalls ;
extern   unsigned  stack[];
extern   int      st_index ;    /* Stack size.                     */

void     decompose(unsigned );
void     adjust_output( uchar *, uchar far *, unsigned, unsigned * );

/*====================== compress() ============================*/
compress( compress_io,compress_work, ptr_bufr_size )
uchar           *compress_io,far *compress_work ;
unsigned        *ptr_bufr_size ;

{
        uchar     *input;
        unsigned  far    *output;
        char      *ptr_new_output;
        unsigned  bufr_size;
        unsigned  code ;
        register  unsigned      data_index=0  ;
        unsigned  out_index=0 ;
        struct    word  {
                        uchar   first;
                        uchar   second;
```

```
                } Liword, Ljword;
unsigned  Li, Lj ,old_Lj ;
unsigned  longblk;
int       bigstk ;
unsigned  position, indexl ;
register  unsigned      j ;

input=compress_io;
output=(unsigned far *)compress_work;
                        /* Li = first input element.       */
Li= input[data_index++] ;
output[out_index++] = Li;
Liword.first=Li;
                        /* Lj = second input element.      */
Lj = input[data_index++] ;
Liword.second = Lj;
Ljword.first = Lj;
bufr_size=*ptr_bufr_size;       /* Find bufr_size.         */
                        /* Loop while there is more input. */
while( data_index <  bufr_size )
        {
        Ljword.second= input[data_index];
        position = 256; /* Start searching after 256.      */
        longblk = Lj;
        bigstk  = -1 ;
        while( data_index < bufr_size )
                {
                if( scan_w4(Ljword, &code, position) )
                    {
                        /* st_index  points  to  the  last */
                        /* element on stack.               */
                    decompose( code ) ;
                    indexl = data_index ;
                    if( ((bufr_size - indexl) >= st_index ) &
                                ( st_index > bigstk ) )
                        {
                        for(j=1;(j <= st_index) &
                              (input[indexl++]==stack[j]);)
                            {
                            j++;
                            }
                        if( j == (st_index+1) )
                                {
                                bigstk = st_index ;
                                longblk = code ;
                                }
                        }
                        position = code + 1;
                    }
                 else
```

```
              break ;
            }
        old_Lj=Lj;
        if( Lj == longblk )
             ;
        else
            {
            Lj = longblk ;
            data_index += bigstk ;
            }
        output[out_index++] = Lj;
                /* If the tables are not full yet, */
        if(next_code<MAX_SIZE)
                /* then string --> string table,   */
                /* i.e put w and k in the wl_table */
                /* and w2_table respectively at the*/
                /* position indexed by next_code.  */
            update_tables( Li, Lj, w3 )
          else
            extracalls++ ;
          Li = Lj ;
          Liword=Ljword;
          Lj = Ljword.first = input[data_index++];
          }
                /* Make sure the last symbol was   */
                /* sent to the output.             */
    if( data_index == bufr_size )
        {
        output[out_index] = input[bufr_size-1] ;
        out_index++;
        }
                  /* Back the output codes from a */
                  /* string of words format to a  */
                  /* string  of  12  bits  codes  */
                  /* format.  The  input  to      */
                  /* adjust_output() is compress_ */
                  /* work. It sends the output in */
                  /* in  the  final  form  in     */
                  /* compress_io.                 */
    adjust_output(compress_io ,compress_work,
                              2*out_index+1, ptr_bufr_size ) ;
}
/*------------------------- END compress() -------------------------*/
/*--------------------------- END Cmprs.c ---------------------------*/
```

21.2. File Tables.c

```
#include        <stdio.h>
#include        <memory.h>
```

```
#include        <malloc.h>
#define         MAX_SIZE        4096
#define         ALPHABET_SIZE   256
#define         uchar           unsigned char

                                   /* Definition of GLOBAL         */
                                   /* variables.                   */
unsigned        w1_table[MAX_SIZE] ;
unsigned        w2_table[MAX_SIZE] ;
unsigned        w4_table[MAX_SIZE] ;
unsigned        *ptr_w1_table=w1_table;
unsigned        *ptr_w2_table=w2_table;
unsigned        *ptr_w4_table=w4_table;
int             next_code ;
unsigned        extracalls=0 ;

/*========================== init_table() =========================*/
/* This function initializes every element in int_table to a com-  */
/* bination that will  never occur. Since the code is only 12 bits */
/* long then the 16 bits used to hold these codes are to be <=     */
/* 0xfff. For this reason in this program the 0xffff code is used  */
/* to solve the above problem. It should be noted that any combin- */
/* ation > 0xfff should work correctly as well. Then the first 256 */
/* symbols in w2_table are initialized to 0-255.                   */
/*=================================================================*/
init_table()

{
        register        int     index ;

                                   /* Set  every  byte  in  the    */
                                   /* int_table  to  0xffff  (i.e. */
                                   /* every code word = 0xffff) so */
                                   /* that no code will match with */
                                   /* it, because the actual codes */
                                   /* are only 12 bits.            */
        memset( (char *) w4_table,0xff,MAX_SIZE*2);
        memset( (char *) w1_table,0xff,MAX_SIZE*2);
                                   /* Set 1st 256 of char_table to */
                                   /* be the extended ASCII codes. */
        for( index=0; index < ALPHABET_SIZE; index++ )
                w2_table[index] =  index ;
        next_code = ALPHABET_SIZE;
}
/*------------------------ END init_table() -----------------------*/
/*-------------------------- END Tables.c -------------------------*/
```

21.3. File Scanw4.asm

```
NAME     SCAN_W4
TITLE    SCANNING OF THE  W4-TABLE
PUBLIC   _scan_w4

LI_WORD  EQU     [BP+4]; PASSED PARAMETERS.
ptr_code EQU     [BP+6]
position EQU     [BP+8]

DGROUP   GROUP  CONST,  _BSS,   _DATA
         ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP

_DATA    SEGMENT
EXTRN    _ptr_w4_table:WORD
EXTRN    _next_code:WORD
_DATA    ENDS

_scan_w4         PROC    NEAR
         PUSH    BP
         MOV     BP,SP
         PUSH    DI
         PUSH    SI
         PUSH    ES
         MOV     AX,DS
         MOV     ES,AX
                                ; INITIALIZE REGISTERS TO THE
                                ; CORRESPONDING PARAMETERS PASSED
         MOV     AX,LI_WORD     ; FROM THE CALLING PROGRAM.
                                ; DI = POINTER TO THE TABLE USED IN
                                ; THE SEARCH. IT HOLDS THE FIRST
                                ; AND SECOND CHARACTERS FOR
                                ; EACH CODE.
         MOV     DI,_ptr_w4_table
         MOV     BX,position
         SHL     BX,1
         ADD     DI,BX
         MOV     CX,_next_code  ; CX = NEXT NUMBER NOT USED IN THE
                                ; TABLES YET.
         SUB     CX,position
         JZ      NOMATCH
LOOP1:                          ; SCAN THE WORD TABLE STARTING FROM
         REPNE   SCASW          ; DI UP TO CX ELEMENTS. BIT ZERO IS
                                ; ZERO. IF ZF= 0 WE FINISHED THE SCAN
         JNE     NOMATCH        ; BEFORE ANY MATCH. SO GO TO NOMATCH.
NOMATCH:
         MOV     AX,0           ; NO MATCH SO RETURN ZERO IN AX
         JMP     SCAN_DONE      ; SCAN IS DONE.
MATCH:                          ; THERE WAS A MATCH SO STORE 1 IN
                                ; FOUND.
```

```
                                  ; MAKE DI = LENGTH OF SCANNED WORDS.
         SUB      DI,_ptr_w4_table
         SHR      DI,1            ; MAKE DI = NUMBER OF SCANNED WORDS.
         DEC      DI              ; ADJUST FOR THE EFFECT OF THE ONE
                                  ; MORE WORD LOOP STEPPING.
                                  ; SCAN WILL RETURN AX = CODE = NUMBER
                                  ; OF WORDS SCANNED TILL WE FOUND A
                                  ; MATCH (i.e. INDEX OF THE MATCHED
                                  ; ELEMENT IN EITHER TABLE) .
         MOV      BX,ptr_code
         MOV      [BX],DI
         MOV      AX,1
SCAN_DONE:
         POP      ES
         POP      SI
         POP      DI
         MOV      SP,BP
         POP      BP
         RET
_scan_w4          ENDP
_TEXT             ENDS
END
/*------------------------ END Scanw4.asm -------------------------*/
```

22. APPENDIX K. TABLE USED IN METHOD LZWB-2

Table 22.1. Extended LZW tables to be used with method LZWB2-B

Symbol	String	w	k
256	01	0	128
257	011	0	129
258	0111	0	130
259	01111	0	131
260	011111	0	132
261	0111111	0	133
262	10	128	0
263	110	129	0
264	1110	130	0
265	11110	131	0
266	111110	132	0
267	1111110	133	0
268	001	1	128
269	0011	1	129
270	00111	1	130
271	001111	1	131
272	0011111	1	132
273	00111111	1	133
274	100	128	1
275	1100	129	1
276	11100	130	1
277	111100	131	1
278	1111100	132	1
279	11111100	133	1
280	0001	2	128
281	00011	2	129
282	000111	2	130
283	0001111	2	131
284	00011111	2	132
285	000111111	2	133
286	1000	128	2
287	11000	129	2
288	111000	130	2
289	1111000	131	2
290	11111000	132	2
291	111111000	133	2
292	00001	3	128
293	000011	3	129
294	0000111	3	130
295	00001111	3	131
296	000011111	3	132
297	0000111111	3	133
298	10000	128	3

Table 22.1. (Continued)

Symbol	String	w	k
299	110000	129	3
300	1110000	130	3
301	11110000	131	3
302	111110000	132	3
303	1111110000	133	3
304	000001	4	128
305	0000011	4	129
306	00000111	4	130
307	000001111	4	131
308	0000011111	4	132
309	00000111111	4	133
310	100000	128	4
311	1100000	129	4
312	11100000	130	4
313	111100000	131	4
314	1111100000	132	4
315	11111100000	133	4
316	0000001	5	128
317	00000011	5	129
318	000000111	5	130
319	0000001111	5	131
320	00000011111	5	132
321	000000111111	5	133
322	1000000	128	5
323	11000000	129	5
324	111000000	130	5
325	1111000000	131	5
326	11111000000	132	5
327	111111000000	133	5
328	00000001	6	128
329	000000011	6	129
330	0000000111	6	130
331	00000001111	6	131
332	000000011111	6	132
333	0000000111111	6	133
334	10000000	128	6
335	110000000	129	6
336	1110000000	130	6
337	11110000000	131	6
338	111110000000	132	6
339	1111110000000	133	6
340	010	256	0
341	0100	256	1
342	01000	256	2

Table 22.1. (Continued)

Symbol	String	w	k
343	010000	256	3
344	0100000	256	4
345	01000000	256	5
346	010000000	256	6
347	0010	268	0
348	00100	268	1
349	001000	268	2
350	0010000	268	3
351	00100000	268	4
352	001000000	268	5
353	0010000000	268	6
354	00010	280	0
355	000100	280	1
356	0001000	280	2
357	00010000	280	3
358	000100000	280	4
359	0001000000	280	5
360	00010000000	280	6
361	000010	292	0
362	0000100	292	1
363	00001000	292	2
364	000010000	292	3
365	0000100000	292	4
366	00001000000	292	5
367	000010000000	292	6
368	0000010	304	0
369	00000100	304	1
370	000001000	304	2
371	0000010000	304	3
372	00000100000	304	4
373	000001000000	304	5
374	0000010000000	304	6
375	00000010	316	0
376	000000100	316	1
377	0000001000	316	2
378	00000010000	316	3
379	000000100000	316	4
380	0000001000000	316	5
381	00000010000000	316	6
382	000000010	328	0
383	0000000100	328	1
384	00000001000	328	2
385	000000010000	328	3
386	0000000100000	328	4

Table 22.1. (Continued)

Symbol	String	W	k
387	00000001000000	328	5
388	000000010000000	328	6
389	0110	257	0
390	01100	257	1
391	011000	257	2
392	0110000	257	3
393	01100000	257	4
394	011000000	257	5
395	0110000000	257	6
396	00110	269	0
397	001100	269	1
398	0011000	269	2
399	00110000	269	3
400	001100000	269	4
401	0011000000	269	5
402	00110000000	269	6
403	000110	281	0
404	0001100	281	1
405	00011000	281	2
406	000110000	281	3
407	0001100000	281	4
408	00011000000	281	5
409	000110000000	281	6
410	0000110	293	0
411	00001100	293	1
412	000011000	293	2
413	0000110000	293	3
414	00001100000	293	4
415	000011000000	293	5
416	0000110000000	293	6
417	00000110	305	0
418	000001100	305	1
419	0000011000	305	2
420	00000110000	305	3
421	000001100000	305	4
422	0000011000000	305	5
423	00000110000000	305	6
424	000000110	317	0
425	0000001100	317	1
426	00000011000	317	2
427	000000110000	317	3
428	0000001100000	317	4
429	00000011000000	317	5
430	000000110000000	317	6

Table 22.1. (Continued)

Symbol	String	w	k
431	0000000110	329	0
432	00000001100	329	1
433	000000011000	329	2
434	0000000110000	329	3
435	00000001100000	329	4
436	000000011000000	329	5
437	0000000110000000	329	6
438	01110	258	0
439	011100	258	1
440	0111000	258	2
441	01110000	258	3
442	011100000	258	4
443	0111000000	258	5
444	01110000000	258	6
445	001110	270	0
446	0011100	270	1
447	00111000	270	2
448	001110000	270	3
449	0011100000	270	4
450	00111000000	270	5
451	001110000000	270	6
452	0001110	282	0
453	00011100	282	1
454	000111000	282	2
455	0001110000	282	3
456	00011100000	282	4
457	000111000000	282	5
458	0001110000000	282	6
459	00001110	294	0
460	000011100	294	1
461	0000111000	294	2
462	00001110000	294	3
463	000011100000	294	4
464	0000111000000	294	5
465	00001110000000	294	6
466	000001110	306	0
467	0000011100	306	1
468	00000111000	306	2
469	000001110000	306	3
470	0000011100000	306	4
471	00000111000000	306	5
472	000001110000000	306	6
473	0000001110	318	0
474	00000011100	318	1

Table 22.1. (Continued)

Symbol	String	w	k
475	000000111000	318	2
476	0000001110000	318	3
477	00000011100000	318	4
478	000000111000000	318	5
479	0000001110000000	318	6
480	00000001110	330	0
481	000000011100	330	1
482	0000000111000	330	2
483	00000001110000	330	3
484	000000011100000	330	4
485	0000000111000000	330	5
486	00000001110000000	330	6
487	011110	259	0
488	0111100	259	1
489	01111000	259	2
490	011110000	259	3
491	0111100000	259	4
492	01111000000	259	5
493	011110000000	259	6
494	0011110	271	0
495	00111100	271	1
496	001111000	271	2
497	0011110000	271	3
498	00111100000	271	4
499	001111000000	271	5
500	0011110000000	271	6
501	00011110	283	0
502	000111100	283	1
503	0001111000	283	2
504	00011110000	283	3
505	000111100000	283	4
506	0001111000000	283	5
507	00011110000000	283	6
508	000011110	295	0
509	0000111100	295	1
510	00001111000	295	2
511	000011110000	295	3
512	0000111100000	295	4
513	00001111000000	295	5
514	000011110000000	295	6
515	0000011110	307	0
516	00000111100	307	1
517	000001111000	307	2
518	0000011110000	307	3

Table 22.1. (Continued)

Symbol	String	w	k
519	00000111100000	307	4
520	000001111000000	307	5
521	0000011110000000	307	6
522	00000011110	319	0
523	000000111100	319	1
524	0000001111000	319	2
525	00000011110000	319	3
526	000000111100000	319	4
527	0000001111000000	319	5
528	00000011110000000	319	6
529	000000011110	331	0
530	0000000111100	331	1
531	00000001111000	331	2
532	000000011110000	331	3
533	0000000111100000	331	4
534	00000001111000000	331	5
535	000000011110000000	331	6
536	000000001	7	128
537	0000000011	7	129
538	00000000111	7	130
539	000000001111	7	131
540	0000000011111	7	132
541	00000000111111	7	133
542	100000000	128	7
543	1100000000	129	7
544	11100000000	130	7
545	111100000000	131	7
546	1111100000000	132	7
547	11111100000000	133	7
548	1000000001	542	128
549	10000000011	542	129
550	100000000111	542	130
551	1000000001111	542	131
552	11000000001	543	128
553	110000000011	543	129
554	1100000000111	543	130
555	11000000001111	543	131
556	111000000001	544	128
557	1110000000011	544	129
558	11100000000111	544	130
559	111000000001111	544	131
560	1111000000001	545	128
561	11110000000011	545	129
562	111100000000111	545	130

Table 22.1. (Continued)

Symbol	String	w	k
563	1111000000001111	545	131
564	0000000001	8	128
565	00000000011	8	129
566	000000000111	8	130
567	0000000001111	8	131
568	00000000011111	8	132
569	000000000111111	8	133
570	1000000000	128	8
571	11000000000	129	8
572	111000000000	130	8
573	1111000000000	131	8
574	11111000000000	132	8
575	111111000000000	133	8
576	10000000001	570	128
577	100000000011	570	129
578	1000000000111	570	130
579	10000000001111	570	131
580	110000000001	571	128
581	1100000000011	571	129
582	11000000000111	571	130
583	110000000001111	571	131
584	1110000000001	572	128
585	11100000000011	572	129
586	111000000000111	572	130
587	1110000000001111	572	131
588	11110000000001	573	128
589	111100000000011	573	129
590	1111000000000111	573	130
591	11110000000001111	573	131
592	00000000001	9	128
593	000000000011	9	129
594	0000000000111	9	130
595	00000000001111	9	131
596	000000000011111	9	132
597	0000000000111111	9	133
598	10000000000	128	9
599	110000000000	129	9
600	1110000000000	130	9
601	11110000000000	131	9
602	111110000000000	132	9
603	1111110000000000	133	9
604	100000000001	598	128
605	1000000000011	598	129
606	10000000000111	598	130

Table 22.1. (Continued)

Symbol	String	w	k
607	100000000001111	598	131
608	1100000000001	599	128
609	11000000000011	599	129
610	110000000000111	599	130
611	1100000000001111	599	131
612	11100000000001	600	128
613	111000000000011	600	129
614	1110000000000111	600	130
615	11100000000001111	600	131
616	111100000000001	601	128
617	1111000000000011	601	129
618	11110000000000111	601	130
619	111100000000001111	601	131
620	000000000001	10	128
621	0000000000011	10	129
622	00000000000111	10	130
623	000000000001111	10	131
624	0000000000011111	10	132
625	00000000000111111	10	133
626	100000000000	128	10
627	1100000000000	129	10
628	11100000000000	130	10
629	111100000000000	131	10
630	1111100000000000	132	10
631	11111100000000000	133	10
632	1000000000001	626	128
633	10000000000011	626	129
634	100000000000111	626	130
635	1000000000001111	626	131
636	11000000000001	627	128
637	110000000000011	627	129
638	1100000000000111	627	130
639	11000000000001111	627	131
640	111000000000001	628	128
641	1110000000000011	628	129
642	11100000000000111	628	130
643	111000000000001111	628	131
644	1111000000000001	629	128
645	11110000000000011	629	129
646	111100000000000111	629	130
647	11110000000000011111	629	131