DESIGN AND IMPLEMENTATION OF A PC BASED MEDICAL IMAGE WORKSTATION

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Ömer Nezih Gerek

July 1993

DESIGN AND IMPLEMENTATION OF A PC BASED MEDICAL IMAGE WORKSTATION

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND

ELECTRONICS ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCES

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Ömer Nezih Gerek

July 1993

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. A. Enis Çetin(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Hayrettin Köymen

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Haldun M. Özaktaş

Approved for the Institute of Engineering and Sciences:

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Sciences

ABSTRACT

DESIGN AND IMPLEMENTATION OF A PC BASED MEDICAL IMAGE WORKSTATION

Ömer Nezih Gerek
M.S. in Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. A. Enis Çetin
July 1993

In this thesis, the implementation of a compact medical image workstation for radiological image processing is described. The workstation is desired to contain sufficient amount of utilities for medical image displaying purposes. Furthermore, it should be affordable for a practicing radiologist. Because of this, the well standardized and cheap PC (Personal computer) media is selected for the workstation machine. In order to handle large amount of digital image data, image compression techniques are studied and an implementation of the Joint Photographic Experts Group (JPEG) standard is used. Furthermore, a class of transform techniques, namely the Block Wavelet Transform (BWT) is experimented for substituting the role of the Discrete Cosine Transform (DCT). The peripheral devices such as a gray tone scanner and high capacity optical discs are then integrated to the computer by the developed software.

Keywords : Medical imaging, PC, transform techniques, JPEG, BWT, peripheral devices.

ÖZET

Ömer Nezih Gerek
Elektrik ve Elektronik Mühendisliği Bölümü Yüksek Lisans
Tez Yöneticisi: Doç. Dr. Dr. A. Enis Çetin
Temmuz 1993

Bu tezde, radyoloji görüntülerinin işleneceği bir tıbbi görüntü işleme istasyonunun gerçeklenmesi açıklanmaktadır. Bu iş istasyonunun tıbbi görüntü işlenmesinde yeterli miktarda unsuru kapsaması istenmektedir. Bunun da ötesinde, ortaya çıkan istasyonun çalışmakta olan bir radyolog tarafından alınabilir ucuzlukta olması gerekmektedir. Bu sebeple, standart olmuş ucuz kişisel bilgisayar ortamı cihaz için uygun görülmüştür. Büyük miktarda sayısal görüntü bilgisini saklamak amacıyla görüntü sıkıştırma yöntemleri incelenmiş ve JPEG standartının gerçeklenmesi olan bir yazılım kullanılmıştır. Bununla beraber, yeni bir sınıf dönüşüm yöntemi olan Sayısal Dalgacık Dönüşümü (SDD), Sayısal Kosinüs Dönüşümünün (SKD) yerinde kullanılabilecek şekilde denenmiştir. Gri ton tarayıcı ve yüksek kapasiteli optik disk türü çevresel cihazlar geliştirilen yazılımla daha sonra bu kişisel bilgisayara eklenmiştir.

Anahtar Kelimeler : Tıbbi görüntüleme, kişisel bilgisyar, JPEG, Sayısal Dalgacık Dönüşümü, çevresel cihazlar.

ACKNOWLEDGEMENT

I would like to thank to Dr. A. Enis Çetin for his supervision, guidance, suggestions and encouragement through the development of this thesis.

I would also like to thank to Dr. Hayrettin Köymen and Dr. Haldun Özaktaş for reading and commenting on the thesis.

I want to express my special thanks to Ogan Ocalı for his help in developing the software.

It is a pleasure to express my thanks to all my friends for their valuable discussions and helps, and to my family for their encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

Computer aided storage and display has become an essential tool for medical imaging. This is basically because of the huge growth in the number of medical images with the progress of medical knowledge and technology. Many acquisition, storage, display and manipulation devices became available with their supporting softwares for the last ten years, however the needs and requirements are not very well defined in this area. Because of this, lots of non-standard and independent softwares have been developed for different platforms.

The purpose of this thesis is to build up a Medical Image Workstation together with the necessary software and peripheral devices. The utilities of the Medical Image Workstation should be sufficient for a practicing radiologist.

In this chapter,the description of the system and a brief layout is presented. Furthermore, basic restrictions of the system in terms of software developments are studied. There is also a comparison of our system with the present published medical imaging systems in this chapter. The rest of the thesis will be about the detailed parts of the developed software and its interface with the peripherals. This thesis is composed of three main parts and so is the developed software. The parts of the software are *Image display and manipulation*, [1], *Data-base management and storage* [1] - [4], and *experimental observations about the system* [1]. Basically, the first two are the necessary items for a medical image workstation. The display and data manipulation software will be investigated in *Chapter* 2, and *Chapter* 3 will concern about data and record storage. In chapter 3, experimental results and installation of the peripherals and necessary programs will be investigated. In the last chapter, a total overview of the system and some proposals of future work on

this subject will be presented. An overview of image file formats and image compression techniques will be given in appendixes.

1.1 System limitations

Technically, a Medical Picture Archiving and Communication System (PACS) [3] consists of a number of components which are image acquisition, communications, archiving, display, image processing, and human-machine interface. A reasonable PACS have a file server computer which has a large storage capacity and large memory to cope with the requests of the connected computers [5]. The network must also have high capacity to enable large amount of data flow. This kind of centralized storage has lots of advantages such as the flexibility of increasing the workstation number without any effort on considering the image manipulation and storage for individual computers. Furthermore, many other image filing systems can be connected to the PACS centralized storage [3] system. This approach is suitable for large and relatively expensive medical imaging systems. However, we have no way to experiment this situation. This system requires some number of connected computers which should be of different operating systems to test the protocol flexibility. The implementation of this configuration can be done at a good organized radiology department, but this kind of a work is beyond the purposes of this thesis. Another point about a medical PACS is the difficulty in assembling all required components. A large PACS requires the joint knowledge of many different people from different backgrounds to cover all aspects of the system. Economically, both the implementation and the maintenance of PACS is expensive. The implementation requires research and time. In terms of maintenance, one should have to consider the adaptiveness of everything. As an example, if the image file formats are changed, the part of the software which recognizes the image data should be easily modified. A software engineer can handle these situations by organizing the program properly and make it as modular as possible.

Our aim in this work is different than the described system. The scope of this thesis is to realize a compact and an affordable workstation.

For our specific case, an important system limitation is about the display hardware. The Personal Computer (PC) [4] displays with the regular graphic adapters have a maximum of 1024×760 resolution with the pixel depth of 8 bits. This is a low definition display, and furthermore, the programming language libraries are insufficient even for handling these levels. The Video Graphics

Arrays (VGAs) have a limited number of register bits (specifically 6) assigned for color components. As given on page 7 in [1], there are 64 different choices of levels from the three color components; Red, Green and Blue. This results with obtaining 64x64x64 different colors, however the same card allows the display of at most 256 unique colors of these thousands of colors.

When a monochrome monitor is considered, the situation is even worse. We know (and practically tested) that a monochrome monitor takes the three color components, calculates their exact average and prints that pixel to the screen. In this way, one can easily show that the number of gray tones that can be obtained is 64 + 64 + 64 = 192. Unfortunately none of the standard standard graphic libraries of any programming language and of any compiler producing company supports the utilization of 192 gray tone display on a PC.

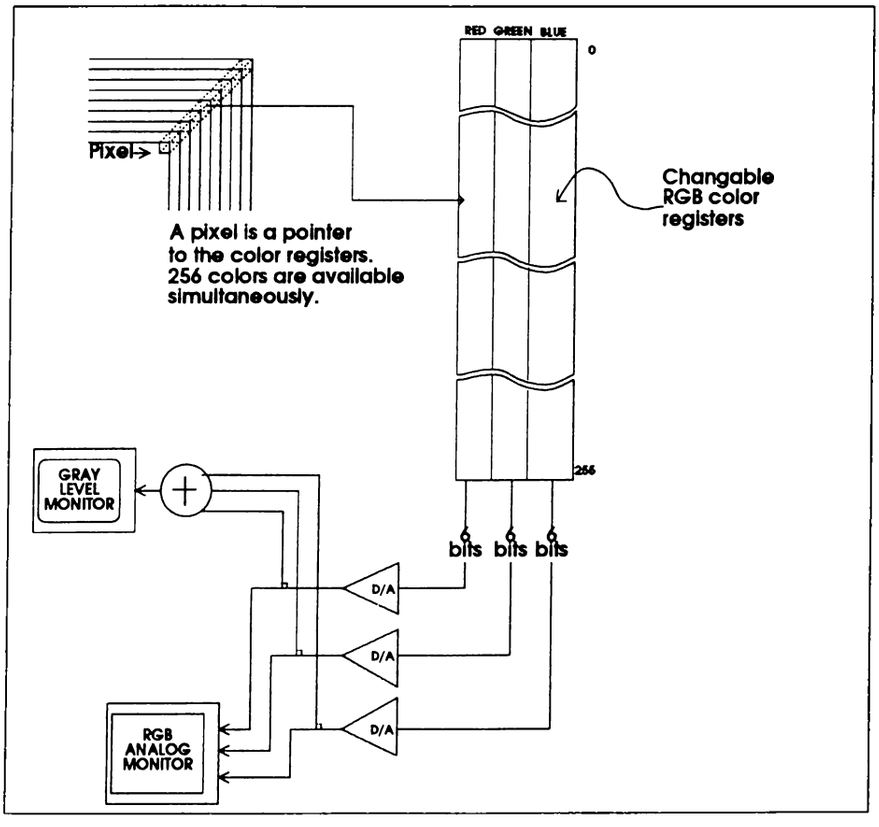


Figure 1.1: Color map installation

The basic problem here is about the standards. The highest resolution the VGA standard supports with 256 colors is only 320×200. Furthermore, 256 colors correspond to 64 tones of gray in the monochrome case. This restriction is because of the small graphic display memory that was available when VGA had became a standard. After the emergence of cheap memory and its application to display technology, every company made its own Super Video Graphics

Array (SVGA). Each such SVGA card has a standard part of the VGA and a memory place which can handle higher resolutions. The high memory address is the nonstandard place. A compiler should obviously fail to cope with all the memory addresses that different companies propose. As an example, the SVGA that we use is the one produced by *OAK Technologies Inc.* For the resolution of 800×600 with 256 colors it has a video frame buffer address starting from A000 and mode number 54h. We wrote the frame buffer addressing, color map installation and put and get point operations in the low level, i.e. assembly language. In order to obtain the interrupt addresses of 800×600×256 modes corresponding to other SVGA cards, we developed a program depending on the public domain software by Kendall Bennett. That software could figure out the name of the SVGA card installed on the computer, the corresponding interrupt addresses of all present modes and the available memory on the frame buffer. Our program communicates with the modified version of that public domain program and gets the interrupt address of the installed VGA. In this way, we are able to handle the video cards corresponding to most of the known SVGA producers.

Another important limitation directly comes from the operating system of the PC namely "Disk Operating System" (DOS) [7]. Because of the early standardization of DOS, there are very strict memory limitations. It is well known that the main memory area is restricted to 640 K bytes which is normally partitioned to 64 K segments for efficient use of memory. When the program and data size exceeds 64 K, a C user has to change the memory model used in the compiler. Specifically, we used the "large" memory model [8], [9] in the Borland^R C++ compiler. The execution and data access speeds decrease when the memory model increases, but the operating system is still not capable of paging, disk swapping or even extended memory allocation (flat memory). One has to know the exact place of free extended memory in order to use it under DOS. If there are some peripherals using addresses from the extended memory (like the one we have), it is dangerous to make data transfer from extended memory. Without knowing the exact allocated memory locations, there is always a possibility of corrupting the functions of peripherals and other extended memory using programs. In terms of conventional memory, the C under DOS has very safe "alloc" and "free" operations [8], [9]. When the memory is not available at the amount of request, the "alloc" function returns an error without causing any more problems. If DOS is used directly without some other environments such as "WINDOWS^{TM}", it is difficult to be safe under the extended memory. This problem is very disturbing in terms of writing a display software because we cannot read all the large image data as a

2-D array once. This limitation results with the use of disk access all the time during a display operation. For example if the image is scrolled or zoomed, those new parts have to be read from the disk. The use of the display buffer is again not suitable as there are 8 pages of the same memory location on the screen, each of which are 64 K long.

The instruction execution of the PC can also be considered a limitation when compared to mini computers or desktop workstations, however the time required for executing basic instructions like additions, branchings and comparisons is not noticeable in the total time required for displaying an image. This shows that instruction speeds of the PC could be ignored in terms of system limitations. The basic speed limit comes from large memory transfers, mathematical calculations like division or trigonometric operations and operations that require disk access or peripheral device access.

1.2 System layout

Hardware:

A brief description of the system is given in the first part of this chapter. Now, the system components will be introduced separately.

The basic element of the system is the computer, i.e. the PC. The PC that we used has a 80486 microprocessor with a 33MHz clock speed and a total of 4MB ram (1MB conventional 3MB extended) [10]. This 80486 microprocessor has the math co-processor unit in itself, so the multiplications and floating point operations are considerably fast. On the other hand, a PC with a 80386 microprocessor performs as well as the PC that we have because the instruction speed has little effect on the total performance of the system. It is previously explained that the bottlenecks are the memory transfer speed, peripheral access speed, and disk access speed. There is not a noticeable difference between 80386 and 80486 in terms of large memory transfer (a block of 64KB to some other memory location) speed. However, because of the better instruction handling, the operations that require loops are faster on a 80486. As a result of this, only the cases for rapid mouse movements and window drawings make the difference between a 80386 and a 80486 in our system.

Beyond the processor, two important units within the computer in terms of speed are the hard disk and the SVGA card. Since disk access is used each

time the image is redrawn on the screen, a fast disk is essential. The speed of the disk is considered in terms of disk access, file position seek and sequential reading. These operations must all be fast for a good disk performance. Reading 800×600 bytes (480KB) from the file and displaying it on the screen requires about 2.5 seconds with our configuration. The time required for VGA card access and frame buffer allocation on the screen is also included in the given time level.

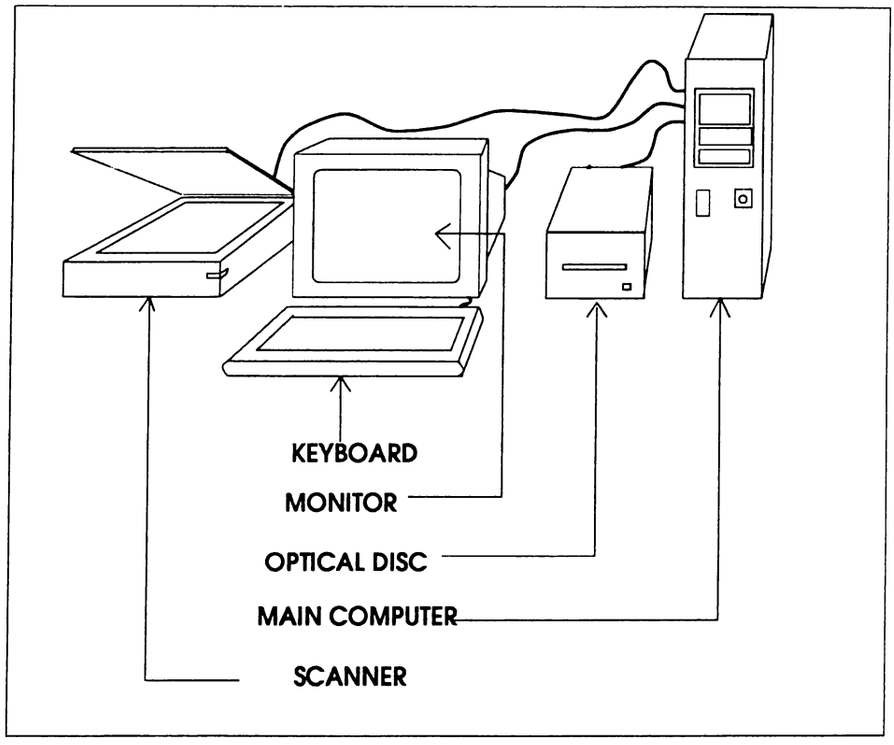


Figure 1.2: System configuration

The importance of the access speed of the SVGA card emerges immediately when the point put and get operations are considered instead of direct video frame buffer access. Most of the VGA cards have similar operation speeds except for very utilized, special purpose video cards (such as the Targa Board). The one we have is one of those regular SVGAs, so its performance could only be increased by making use of the *cache* memory. When some amount of the frame memory is assigned to the cache memory, the put point and get point operations become considerably (2 - 2.5 times) faster. The use of cache memory will also be important when peripheral device access speeds are considered. The interface of the optical disk is SCSI and it also makes use of cache buffering if enabled. It is noticeable that application programs like SMARTDRIVE improves the disk access speed performance by the same cache buffering method.

The gray-tone image scanner is connected to the system by a special purpose extension card inserted in the PC. Converse to the cases of image displaying or data-base management softwares, little effort is spent on the image acquisition software. Essentially, the application software that was supplied with the scanner was easy to use and functional. In this program, the image is previewed first, then the appropriate place of the image is marked. After adjusting the resolution of the scan, the user clicks a mouse button to scan the selected part to disk with a chosen file name. An important fact is that, this program supports many image formats including TIFF, so the image formats are compatible between the display and the acquisition software.

Software:

Sharing the same operating environment is a very important issue in terms of user interface. If the user scans the image from a scanner connected to a workstation with the unix operating system and then does the rest of the processing on a PC, the completeness and ease of use of the system would be disturbed. Even on a PC, we have many versions of DOS and other application environments. In order to avoid system mismatches, we specifically converted the environment of all the programs to the MICROSOFT WINDOWS^{TM}. Since the scanner utility program works under WINDOWS, the other programs adopted to it. This environment is suitable in terms of user interface too. The user easily runs the programs, switches from one application to another and exits from the programs by clicking on a mouse button. This is almost ideal for a non-professional computer user such as a radiologist, whom we suppose is the proposed user of our software.

In order to handle the data-base of the radiology images and the owners corresponding to them, a patient record program is developed. Actually, this is the root of the software in terms of hierarchical interface. When the user executes the "Medical Image Workstation" program by pressing a mouse button, he or she first meets a menu asking for an operation on the patient records. These operations are "insertion of a new record", "seeing the contents of a present record", "directly going to the image viewing menu", "changing the record file name", "creating a record file", and "exit" respectively. The data-base program examines a configuration file to learn the name of the record file. This configuration file should be constructed before running. If the "Medical Image Workstation" program (MIW.EXE) cannot find the configuration file, it assumes a default record file name. This record file is made up of 2000 records each of which has six field items, namely, "name", "age", "ID number", "address", "notes" and "the pointer file for images". There is no way for the

program to work properly without a record file, so if that file does not exist, the user must create one before doing anything else.

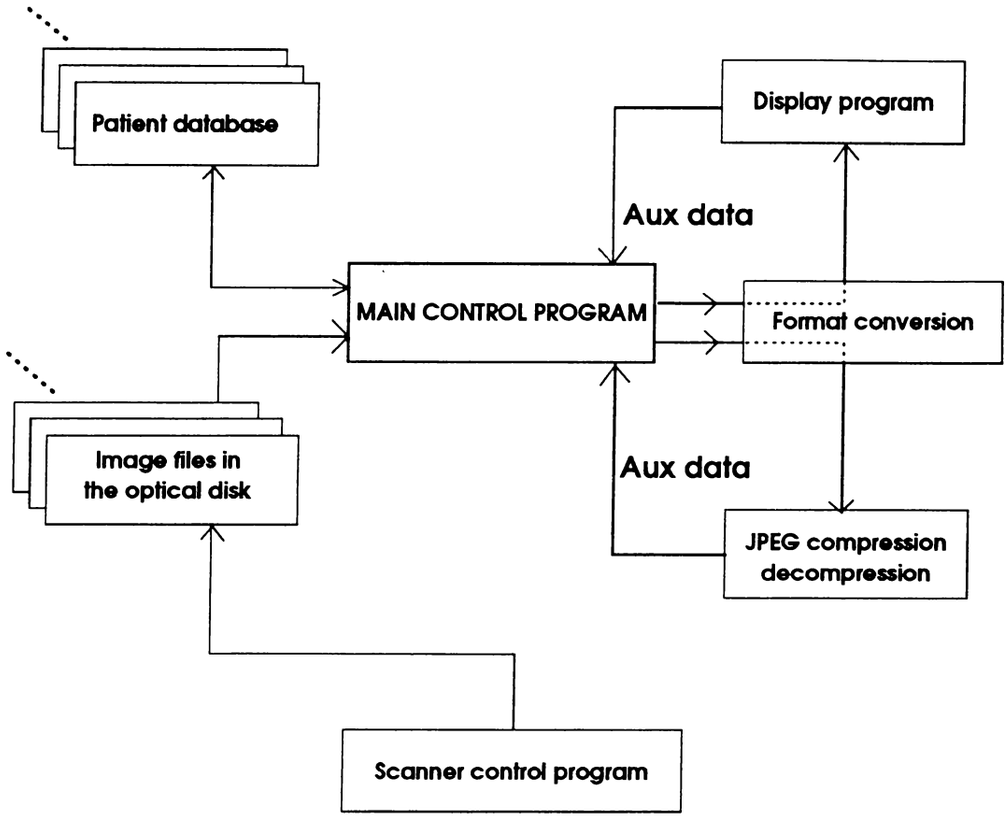


Figure 1.3: Software configuration

When a patient record is written to the file, its place is calculated by a "hash" function applied to the name field. The collision cases are handled in this software and the details will be presented in Chapter 3.

The image viewing and processing part of the software is actually a single program which could be called separately. The integration between the main program and this viewing software is done by executing the viewer from the main program. The data base program has the ability to call the viewer program to avoid exiting each time an image viewing is required. It is noticeable that the integration of the two programs is weak. One should ensure that both of the programs together with the compression program are ready at their proper directories before start.

To compress the images, the an implementation of the Joint Photographic Experts Group (JPEG) standard is executed. This program applies a DCT (discrete cosine transform) based lossy compression on the image. A good feature of this implementation is that the quality - compression ratio trade-off

can be adjusted according to the wills of the user.

The communication between these called programs are done via generating files that are known to all the programs. The necessary data is written in those dummy files. The called program checks those files for information and before termination, it deletes them from the disk. As a result, the user is unaware of these communication files.

Chapter 2

THE IMAGE DISPLAY AND PROCESSING SOFTWARE

The viewing of the radiological images is one of the two basic elements of this thesis. In the first section of this chapter, the communication between the root "data base" program and the image display program is described shortly. Secondly, we describe the directory management of the display software. Finally, the displaying functions and supported image operations are described in detail.

2.1 Root program - display program interface

As mentioned before, the communication between the programs is done by auxiliary files whose names are known to both sides.

One of the record fields is the name of the file which supports the names of the image files corresponding to a patient. This is a text file in which the image file names are written. When the image viewing program "VIEW.EXE" is executed from the root program, it first checks the existence of this "name file". If this file exists, the program takes the file list from the contents of that file, otherwise it behaves as if it is executed from the prompt and displays a menu which asks the commands to be executed regarding the image files on the disk.

On the second step of the display program, the file list is printed on the

screen. After that, the user travels between those file names either by using the mouse or by using the arrow keys. If the "enter" key or left-mouse-button is pressed on a file name, that file is selected and the program displays the image immediately (if there is no error in the image format). There is a possibility that the selected file is in JPEG compressed form. In this case, the display program generates an auxiliary file indicating the the name of the file to be decompressed and terminates. If the F1 key or mid-mouse-button is pressed on a file name and if that file is not a JPEG compressed image, the program generates another auxiliary file to indicate that the file name written is to be compressed, and terminates. Since it is the root program that executes the display program, it goes on running and checks for the image decompression indicator.

If the root program executes the display program after the compression or decompression of an image as indicated above, it generates a different communication file indicating the name of the compressed / decompressed file. The display program goes to the viewing function in the case of decompression or to the list printing function in the case of compression.

There are two other operations done on the image file list, namely "delete from the list" in the patient image file list mode and "add to the list" which is done when all the directory is searched instead of the patient image file list. However, these operations do not require going back and forth between the two programs. They are done totally in the display program.

2.2 Directory management

The issue of changing directories, listing the image files in a directory and selecting the image files is an almost straight-forward but time consuming and difficult job [4], [7], [10]. It requires system operations which usually causes trouble for the DOS systems. A good directory management might be "a good software engineer's job", but we spent little time for smart or understandable programming.

The following is the description of the function "enter_file" called in the display program after checking the auxiliary communication files. Two parameters come in, namely "image_name" and "flag", and three go out, namely "xsize", "ysize" and "header length". If there is a decompressed image file coming in, then "flag" is set accordingly so that there does not occur any file

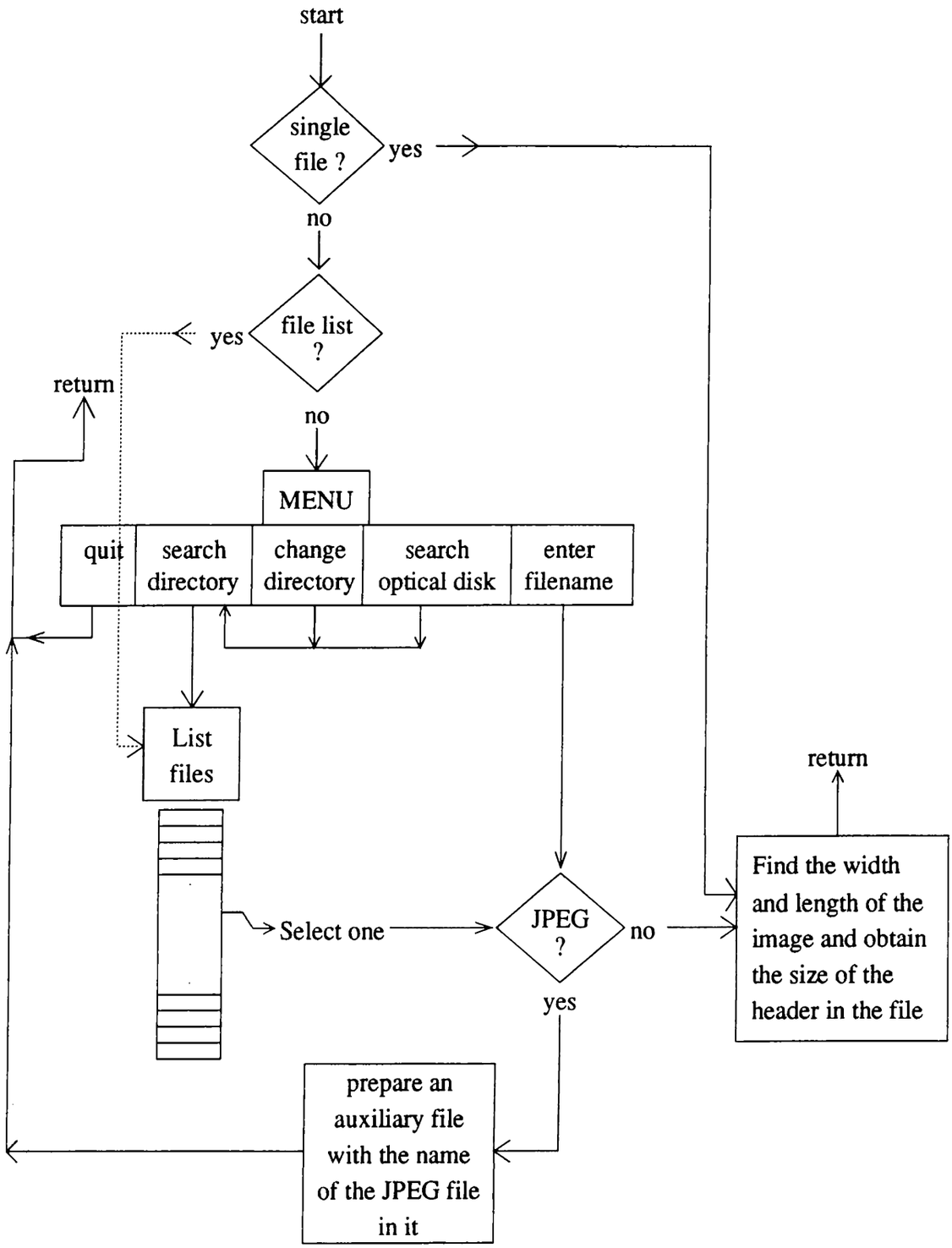


Figure 2.1: flow diagram of the file handling function

list printing to the screen, but that decompressed file is selected immediately. In figure 2.1, this situation is indicated by the "single file" branching option. If the flag has another value, then it indicates that there is an image file list at the incoming file or there is nothing coming in, so the viewer should display a menu to operate.

The width and height, namely "xsize" and "ysize" are obtained from the headers of the files.

The detailed functional description is not given in the figure because there are lots of situations to handle. The mouse interrupts and return values are obtained from the documentation "GMOUSE.DOC" given with the mouse driver and other low level interrupt codes are found from the appendixes of [8], [9].

Upon the call of "enter_file", the above function checks the auxiliary information and selects an image file as described in the figure 2.1. Briefly, we can define the flow of this function in 5 steps :

1) If no specific list
1.1) Select a directory
1.2) Display the contents of the selected directory
2) Else display the contents of the list
3) Wander around the file names by using the mouse
3) Choose a file from the displayed file names
4) Obtain the header information of the chosen file
5) Return

2.3 Color map installation

The issue of displaying relatively high number of gray levels with a high horizontal and vertical resolution is perhaps the most important and contributive part of this thesis. In the described method, a total of 192 gray tones can be obtained and this corresponds to 7.3 bits/pel.

There is a need of modification in the color map for displaying the gray scale images on a B/W screen with a SVGA card [1]. The commercially available gray level display programs assign the same value to the color registers to obtain a gray tone [1], [6], but we used the following method for numbering

the color registers :

Maximum number of colors = 256 : fixed.

R	G	B
0	0	0
0	0	1
0	1	1
1	1	1
1	1	2
1	2	2

In our implementation, we supported a modifiable color map installation in order to be able to make the image brighter, darker, negative, or sharper. During the display mode, the user modifies the color map by pressing "pg-up", "pg-dn", "home", "end" and "insert" keys. Every time these keys are pressed, the following line is executed :
setpalet(setmempalette(map));
with the "map" value determined by the pressed key. The installation of a given color map can also be done in the same way.

The "setmempalette()" function returns a pointer to the calculated look-up table which stands for the color map, and the "setpalet()" function installs that table in the video memory by using the DOS interrupt 10h. The necessary register values before the interrupt is defined as follows :
AX=1012h
BX=0
CX=256
ES=SEGMENT(table)
DX=OFFSET(table)

We assigned six concave up (dark for the normal display) modes indicated by the value of "map" from 1 to 6 with 1 having the highest curvature and 6 having an almost linear curve. The concavity is made by taking the powers of the mapping function with values 4, 3, 2.5, 2, 1.7 and 1.3 and normalizing the values for the maximum to be 255. The value 7 is the linear color map curve and it is the default. From 7 to 13, the curve becomes concave down (the image becomes lighter in the normal display) by taking the powers in the opposite way (0.77, 0.6, 0.5, 0.4, 0.33, 0.25). In order to increase contrast, the

map value is assigned to 14. By pressing the "home" key, the value of "map" is reset to the default value, 7.

The negative display is supported independent of the color map adjustment upon pressing the "end" key. Both in the negative and in the normal display mode, the color map is adjustable in the same way.

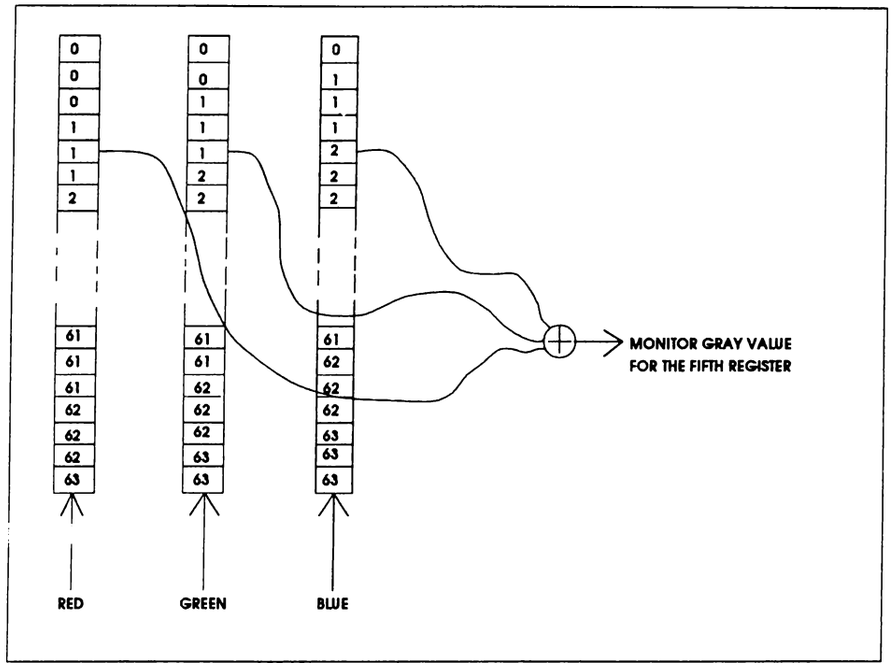


Figure 2.2: Color map rules

The color map editing is not done manually by changing the curve shape arbitrarily. This would require a separate window to manipulate the color map and since DOS itself is not a multi-window environment, there is not much place to display figures. Furthermore, the restoration of the background after closing a window is not done automatically, so we would either consume a lot of memory to keep the background image or we would redraw the image from the disk. Implementing such a feature would not be worth dealing with these two difficulties. Increasing and decreasing brightness, and increasing the contrast are enough for our viewing purposes.

The function "begingraph()" is called to enter the graphics mode. This function assigns the mode value to the AX register and then calls the interrupt 10h. After that, it installs the color map and assigns a pointer ("gpt0") to the frame buffer. The desired graphics mode for resolution and pixel depth is determined by the mode number which is an input variable to "begingraph()". Every SVGA producing company assigns different numbers to different resolution modes. In this program, this function must be called with the specific

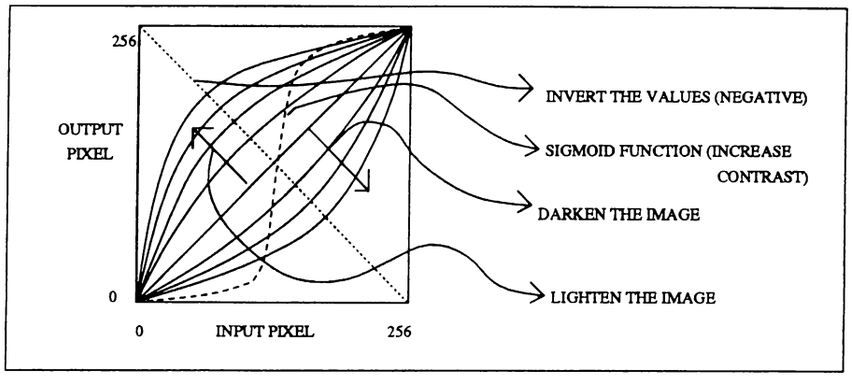


Figure 2.3: Changing the color map

variable so that the SVGA enters to 800x600 mode with 256 colors. This requires a memory limit of at least 512K bytes for the display frame. Since this is the beginning of drawing, the function assigns the default color map (7) for color map installation.

To return to text mode, another function "endgraph()" is used. This function assigns the value "7" to the AX register and calls the DOS interrupt 10h. The value "7" corresponds to the 40x25 text mode with a character cell of 9x16. This is the default monochrome mode at power up.

A normal graphic display does not use our kind of color map installation [1]. The natural way to assign gray tones to 256 colors is to assign equal weighting for all color registers and use the gamma correction. This kind of color map installation results with a total of 64 gray tones and the quantization steps between the gray tones are disturbingly visible for some cases. The following table illustrates the assignments of the color register values with the ordinary method.

Color Register	Red	Green	Blue
0	00	00	00
1	00	00	00
2	00	00	00
3	00	00	00
4	00	00	00
5	01	01	01
6	01	01	01
7	02	02	02
8	03	03	03
9	03	03	03
10	04	04	04
11	05	05	05
.	.	.	.
58	56	56	56
59	58	58	58
60	59	59	59
61	60	60	60
62	61	61	61
63	63	63	63

Calling the begingraph() function with an appropriate parameter is enough for entering the graphics mode. This parameter determines the resolution and number of colors. What is more, it is also possible to change the color map without leaving and re-entering the graphics mode. While the image is being displayed, a function call like "setpalet(setmempalette(7))" results with the loading of the linear color map (the default). In the keystroke control function, we assigned the pg-up and pg-dn keys for incrementing or decrementing the setmempalette() parameter. In the normal mode, pg-up makes the image brighter and the dark tones are separated better, and pg-dn makes the image darker with the opposite effect. By pressing the "home" key, the default '7' parameter is passed and the image becomes normal mapped. To make the image "negative", "end" key must be pressed and to apply a sigmoid mapping for increasing the contrast, "ins" key must be pressed.

2.4 Basic point operations

Two important functions about a graphics software are the "PutPoint" and "GetPoint" functions [1], [6]. The PutPoint function should put a dot with a given given color to the given screen coordinates. Conversely, the GetPoint function should return the color value of the given coordinates on the screen. The implementation of these functions is considerably simple with the known interrupts. Unfortunately, calling interrupt 10h is not a fast operation, so if one wants to display an image on the screen, he or she should use the frame buffer pointer (gpt0) as long as possible. However, during applying some spatial filters, we had to use these functions, so those operations are slow.

For the "PutPoint" function, the the following registers must be set ant the DOS interrupt 10h must be called :
AX = 00FFh & color + 0C00h
BX = 0
CX = Xcoordinate
DX = Ycoordinate

For the "GetPoint" operation, the following registers are set,
AX = 0d00h
CX = Xcoordinate
DX = Ycoordinate,
and the DOS interrupt 10h is called. The pixel value is returned in the AX register

Rectangle drawing and mouse cursor drawing are two important utilities that should be considered separately. When the user moves the mouse, the cursor must move on the screen and when a mouse button is pressed, it must behave accordingly. The left button is assigned to put a point, so that when the mouse is moved with keeping that button pressed, one should be able to put a series of points. On the other hand, the mid button is used for drawing a rectangle to indicate the boundaries of operations like filtering, saving etc. In order to draw a rectangle, one must hold the mid button pressed and move the mouse along the diagonal points of this rectangle. Each time the last corner of the rectangle is changed by moving the mouse, the pixels with positions corresponding to the previous rectangle must be reset to their previous values and

a new rectangle must be drawn at the new place. The necessary functions to perform these operations are "getrect()", "putnewrect()" and "putprevrect()".

All these functions put and get pixels along the lines corresponding to the borders of the window. The functions obtain the borders from the corners of the rectangle which are passed to the functions as a parameter.

The order of call of these functions is as follows :

```
putprevrect(oldx1, oldy1, oldx2, oldy2);
getrect(newx1,newy1,newx2,newy2);
putnewrect(newx1,newy1,newx2,newy2);
oldx1=newx1; oldx2=newx2;
oldy1=newy1; oldy2=newy2;
```

After that, the new corner coordinates (newx2, newy2) are obtained according to the mouse motion.

The motion of the arrow-shaped mouse cursor is handled in a similar manner. When the mouse moves, the old pixel values are restored (a total of twenty pixels) and the new arrow is drawn at the new place.

2.5 Drawing the image

Having defined the frame buffer pointer and the point operations, we can now introduce the image drawing functions. The first of these functions draws the image at its defined size without scaling. If the image size is larger than the screen (800x600), only the fitting part is drawn.

At the beginning of the program, the default setting is such that the up-left corner of the image is at the up-left corner of the screen. When the user presses an arrow key, the image scrolls at the opposite direction at the amount of 40 pixels, but this scrolling is not handled within the drawing functions. When a scroll operation is requested, the main control function simply re-draws the image by passing the horizontal and vertical offsets to the drawing functions as a parameter.

In the first chapter, we had introduced some limitations about the system. Actually, one other limitation is the size of the frame buffer pointer. As a result of being a DOS compatible system, the VGA cards support frame pointers of size 64K bytes. However the size of the screen is obviously more than that, 800 $\times$ 600 with one bytes per pixel is 480K bytes. Even if all the screen were available as a single pointer, we would not be able to allocate all of it at once because that would exceed the conventional DOS memory with the program code and other data in the program. Because of these, redisplaying or scrolling the image requires disk accesses every time.

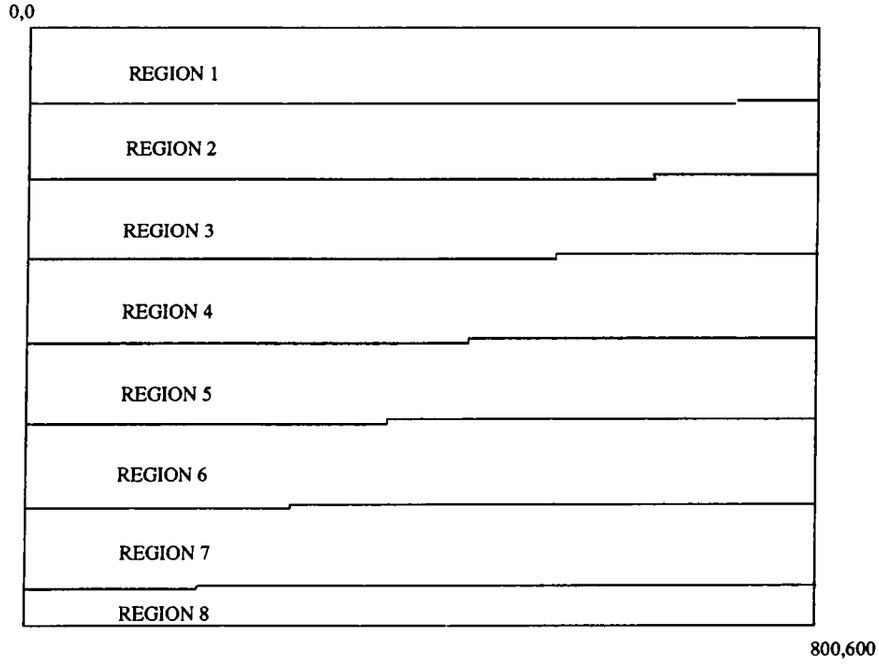


Figure 2.4: Screen partitioning

In order to handle the 64K pointers individually, the programmer must be careful about the points where 64K exactly finishes on the screen. At the beginning, the pointer is at the top of the screen starting from the corner position (0,0), and when the pointer index is incremented, it proceeds like (0,1), (0,2), (0,3),... ,etc. Here the convention is (vertical,horizontal). At the 801st index of the pointer, it switches to the position (1,0). In this way, the first 64K finishes at the point (81,735). The frame buffer pointer must switch to the next memory page which corresponds to the place below the first 64K. The switching between the screen portions can be done by a PutPoint or GetPoint operation inside the desired portion. After one of these operations, the pointer starts to point to the first place of the corresponding portion. Actually, the

DOS addresses of all these pages are the same, but the corresponding Put or GetPoint operations cause to change the memory locations inside the VGA card.

Unfortunately most frame pages start from an almost arbitrary points like (81,735), so special care must be taken about the value of the pointer index where it reaches to the end of the buffer.

All of the eight screen portions are handled similarly. The start positions of these portions are as follows :
(0,0), (81,735), (163,671), (245,607), (327,543), (409,479), (492,415), (574,351)

There are also some other disturbing points to care about. For example the bottom of every image must be cleared after drawing. This is basically because there could be a scroll up operation, so there could remain some drawn pixels at the bottom.

The function described in figure 2.5 draws one pixel per one character in the image file. Although there are 8 portions of 41 lines on the screen, we read half of the data of one portion which corresponds to 21 lines. This is basically because of the memory constraints. If the width of the image is more than 800, we cannot read 41 lines into one array because it exceeds 64K bytes. It is known that the "huge" memory model in the C compilers of the DOS systems support pointers with size larger than 64K bytes, but the memory operations become slow in that case. Because of that, a more practical solution is to stay in a smaller memory model and try to handle pointers with smaller size.

The image borders and the screen borders must be carefully controlled in order to draw the image properly. There are basically two cases to consider for border adjustment (fig. 2.6). In (a), the horizontal data of the image end before the screen border and in (b), there are pixels beyond the screen border. In these situations, the read data must be copied to the frame buffer pointer "gpt0" by taking care of the index increments.

Furthermore, we considered the first and last lines of the screen portions individually in order to handle every kind of unexpected situation (fig. 2.7).

There is also another kind of drawing function which draws the marked region to the screen with magnification. While doing this zoom operation,

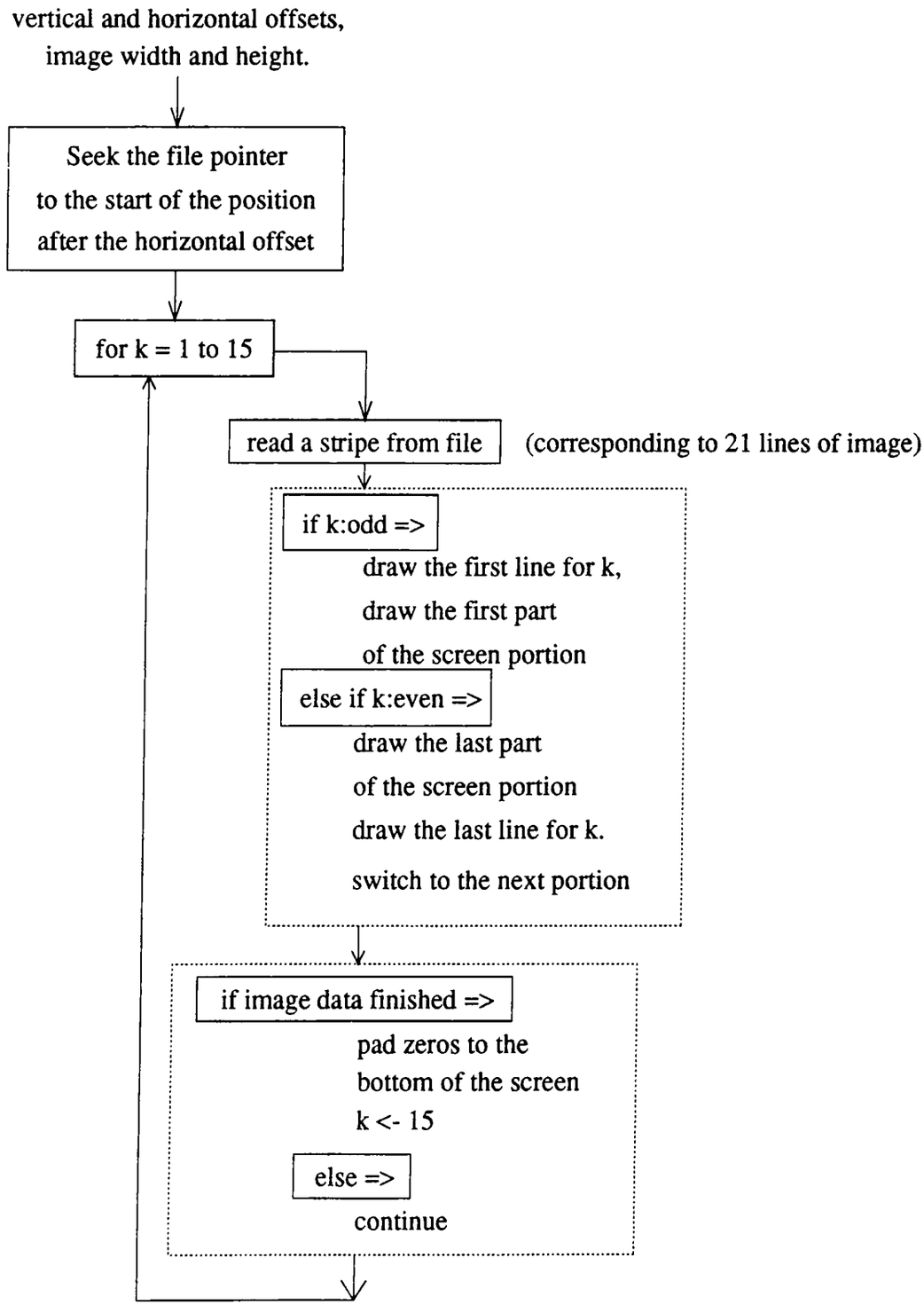
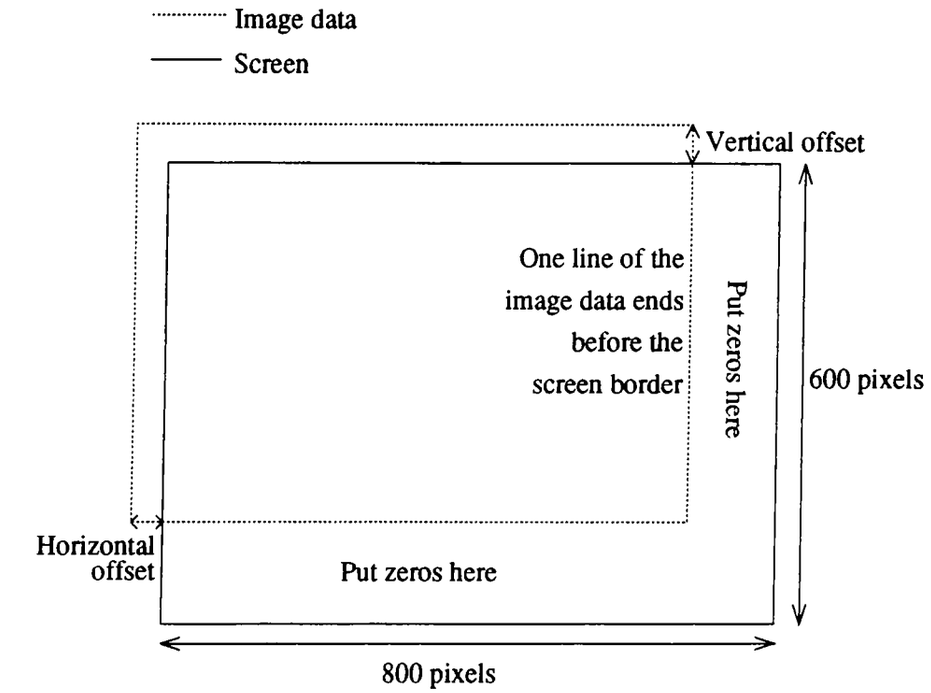
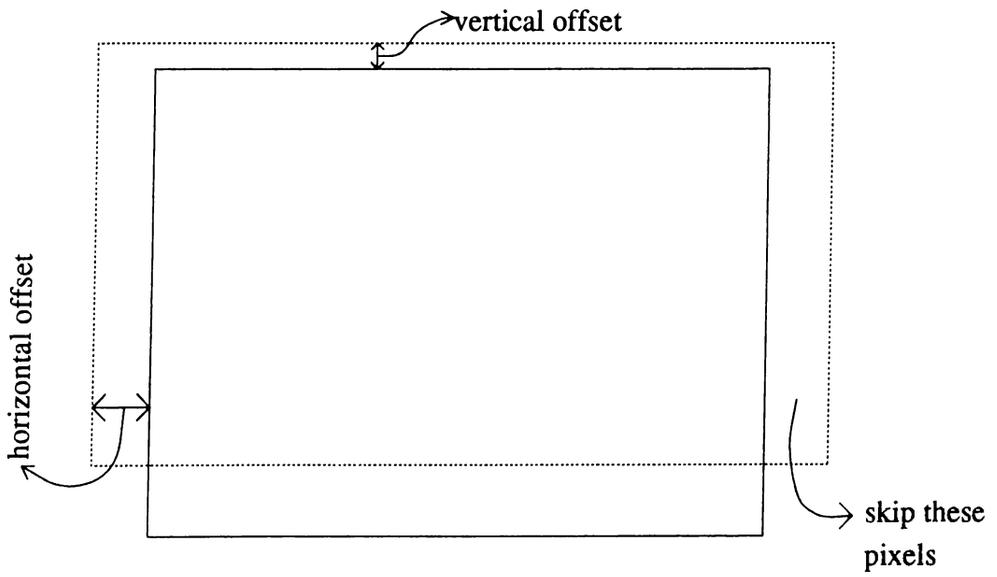


Figure 2.5: drawing the image in normal mode



(a)



(b)

Figure 2.6: two cases for file and screen sizes

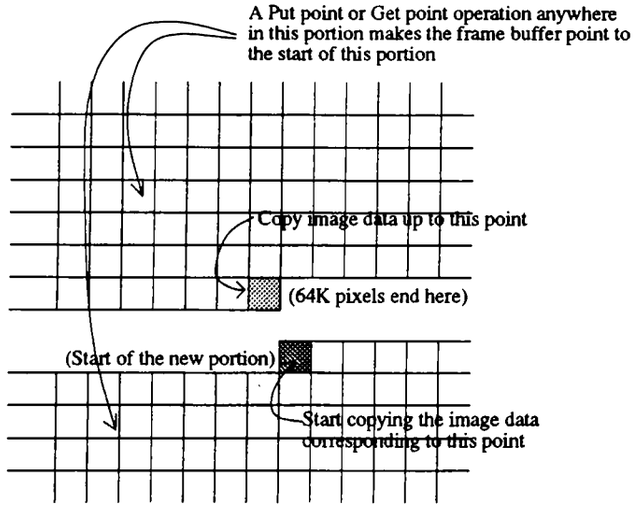


Figure 2.7: the end and start points of screen portions

the aspect ratio of the marking rectangle should be considered, so the function aligns the corner points such that the horizontal size versus vertical size becomes 800 / 600.

A real zooming function should insert necessary amount of zeros between original pixels and then apply a corresponding low pass filter [11], [14]. This is the correct way to do it, but it requires too many computations, so it is extremely slow. In this kind of an application, the user wants to see the result of the operation as soon as possible. Because of this, we preferred to use the simple and fast first-order interpolation. The result is usually satisfactory if one does not apply a zoom with ratio more than three. Figure 2.8 shows the flow diagram of the zoom draw function.

The zoom drawing function is similar to the normal drawing function, but there are more controls. First of all, the corner points of the selected region are modified in order to keep the aspect ratio correct. Other than that, the function reads one line of the image data and makes interpolation calculations on that line and the previous line. The handling of the screen portion switching is done in the same way as in the normal drawing mode, but there are a lot more control statements to consider every kind of situation. The interpolation method is illustrated in figure 2.9. One should note that the fractional pixel positions are rounded to the nearest physical pixel locations and the pixel values are quantized to an integer.

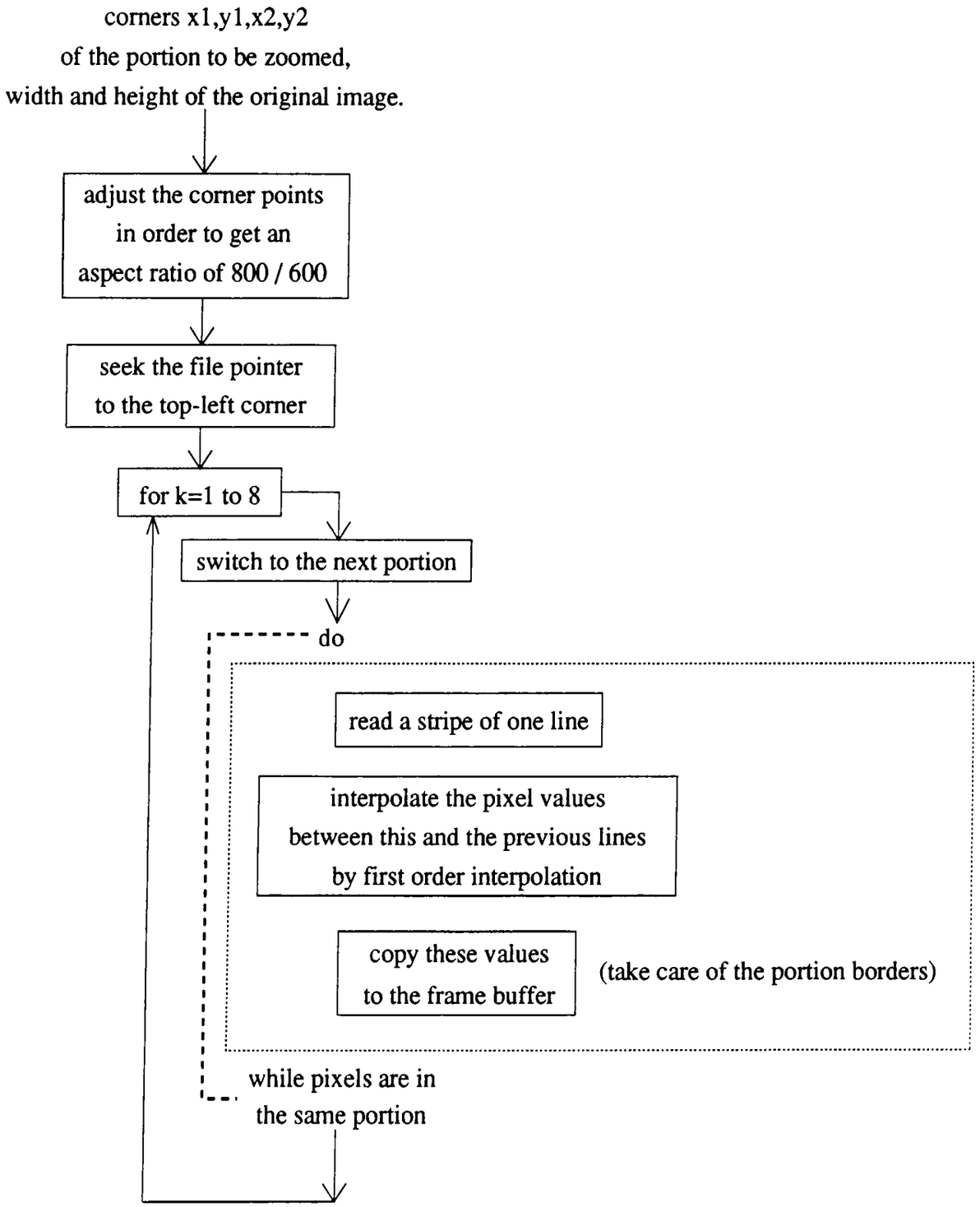


Figure 2.8: drawing the image in zoom mode

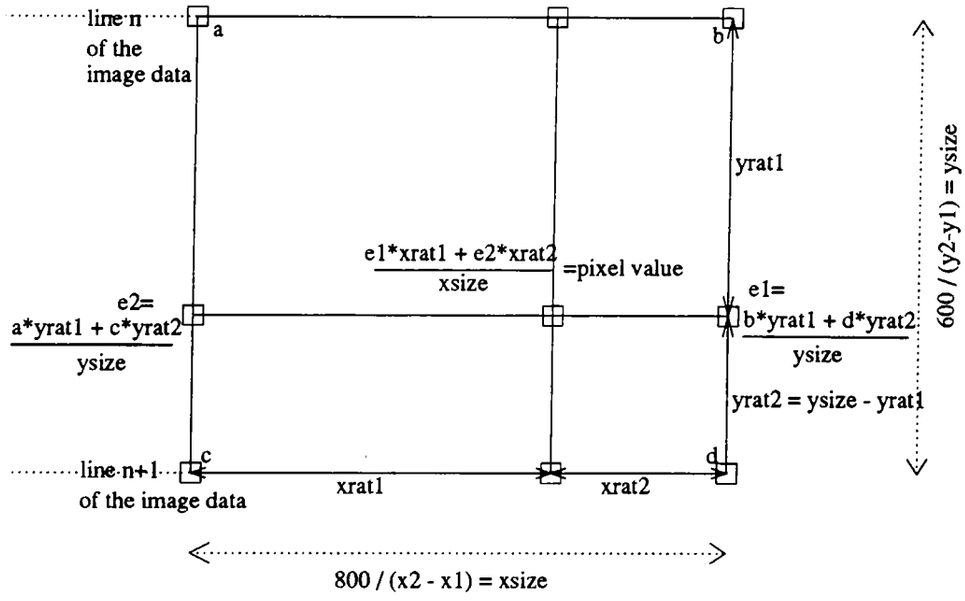


Figure 2.9: first order interpolation

Because of having long source codes and complex control statements, it is better to give the rough idea of how these functions are implemented instead of explaining the details.

There are actually some graphics routines supported with the compilers, but they are not suitable for our purposes. The best a DOS programmer can do with the standard libraries is the 320x200 mode with 256 colors. If a user is satisfied with this resolution, he or she can draw and manipulate images easily. For example the "PutImage" or "GetImage" functions are ready with most of the compilers.

There are also some specific graphics card producers which distribute a powerful library with the necessary hardware. Usually, these kinds of graphics boards are considerably more expansive than the regular SVGAs. An example for this is the TARGA^{TM} board where hundreds of special purpose functions, including displaying, scaling, translation, pixel operations and many more are supported in the distributed software. If this were the case, there would not be a need for developing the preliminary functions for basic operations. Nevertheless, development of these functions is a good exercise about getting into the graphics utilities of computers. The work of this thesis is a chance to see the critical points in low level graphics programming.

2.6 Filtering and other operations

In this section, some basic image processing tools are introduced. These tools include three kinds of low pass filtering, two kinds of high pass filtering, two kinds of median filtering and histogram equalization. All of these filters are applied inside the drawn window and they are 3x3 filters except for histogram equalization. The filter coefficients are calculated according to the rules given in [11] [12] so that the all filters are symmetric and do not cause diffraction-like patterns.

The linear filters essentially apply the following convolution

$$v(m,n) = \sum \sum_{(k,l)\in W} a(k,l)y(m-k,n-l) \tag{2.1}$$

where y(m,n) and v(m,n) are the input and output images respectively, W is the 3x3 window, and a(k,l) are the filter weights as given in figure 2.10.

All the low pass and high pass filters are done in a single function named "smooth()". The corner points of the rectangularly marked region and an indicator variable are passed to this function to perform the convolutional filters. The indicator variable can take five different values corresponding to five different filters. These filter coefficients are defined globally as:

```
                                    /* 3x3 filter coefficients */
float mask[5][3]={ 0.25,0.125,0.0625,          /* Low pass */
                   4.0, -0.5, -0.25,          /* High-pass */
                   0.111111,0.111111,0.111111, /* Averaging */
                   0.0, 0.2, 0.05,   /* Neighbour averaging */
                   7.0, -1.0, -0.5  };  /* Strong high-pass */
/*                  ^    ^     ^
matrix center val___|    |     |
left/right/up/down val___|     |
matrix corner values___________|
*/
```

and according to the indicator, the suitable filter is selected. In every row of the two dimensional array "mask[.][.]", the first value corresponds to the center

of the 3x3 mask, the second value correspond to the neighbors of the center and the last value corresponds to the corners. As a result,
mask[x][0] + 4xmask[x][1] + 4xmask[x][2] = 1
for all filters which is an essential equality for keeping the tone of the image constant.

During applying the 3x3 convolutions, only the necessary amount (3 lines) of the image is taken to the RAM and the operations are performed there. After finishing a line in the selected region, a new line is read from the bottom of the previously read lines. After that the order of the lines are shifted up and the top-most line is thrown away.

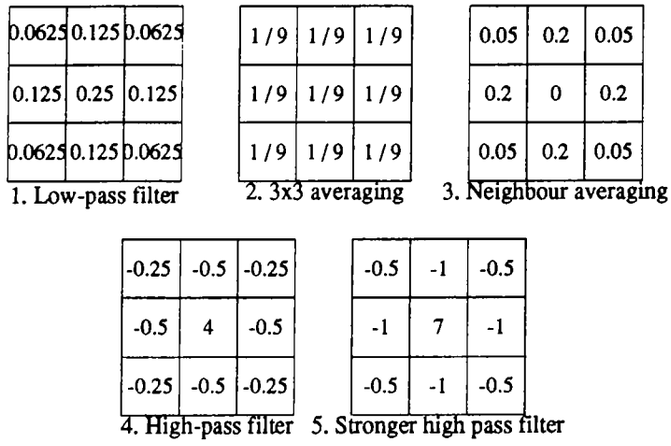


Figure 2.10: Filter masks

Linear filtering operations are necessary in an imaging program because they are the basic tools for image processing [11], [12], [13], [14]. There are a couple of things to do in order to obtain the coefficients of the low-pass and high-pass filters (page 195 in [11]). The first thing is to determine the specifications. For our purposes, the filter size must be 3x3 and it must be zero phase, so that

$$H(\omega_1, \omega_2) = H^*(\omega_1, \omega_2). \tag{2.2}$$

This is equivalent to

$$h(n_1, n_2) = h(-n_1, -n_2) \tag{2.3}$$

for real $h(n_1, n_2)$.

We used the filter design by windowing method in this work. This method first finds the exact desired impulse response, then multiplies that possibly

infinite impulse response sequence with an appropriate window.

$$h(n_1, n_2) = h_d(n_1, n_2)w(n_1, n_2) \tag{2.4}$$

If $h_d(n_1, n_2)$ and $w(n_1, n_2)$ are both symmetric around origin, the resultant filter will also be symmetric.

The matlab FIR2 command uses the hamming window to calculate the finite impulse response coefficients of the specified filter, and we used that command to obtain the filter coefficients given in the program. Here, the hamming window is give by

$$w(t) = \begin{cases} 0.54 + 0.46cos(\pi t/\tau), & if|t| < \tau \\ 0, & otherwise \end{cases} \tag{2.5}$$

for the continuous case. The discrete time case is the time quantized version of this.

Two different kinds of median filtering are implemented in this software [11], [12], [13]. One of them is the regular two dimensional 3x3 median filtering. The other one is applied in a directional manner where the horizontal lines are median filtered in a window of size 3 pixels first, then similar operations are done along the vertical lines in the drawn rectangle window. The regular median filtering causes distortion at the corners of objects in the image, but the directional median conserves those corners. Unfortunately this time the thin lines may possibly vanish. The regular median filter is defined by

$$v(m, n) = median\left\{y(m - k, n - l)\right\}, (k, l) \in W \tag{2.6}$$

where W is a 3x3 window or a 3x1 (or 1x3) line in the case of directional median filtering.

We implemented the median selection of the 9 components in the 3x3 case as follows :

- Open an array of size 256 and reset their values to zero
- For the pixels in the 3x3 window
- Read the value of a pixel and assign to "a"
- Increment the value of the "a"th indexed element in the array
- Get the index of the fifth nonzero element in the array

An example illustration of this is given in figure 2.11. In this example the

fifth nonzero index is "l+1" because the summation of the elements up to and including the l+1st element is five.

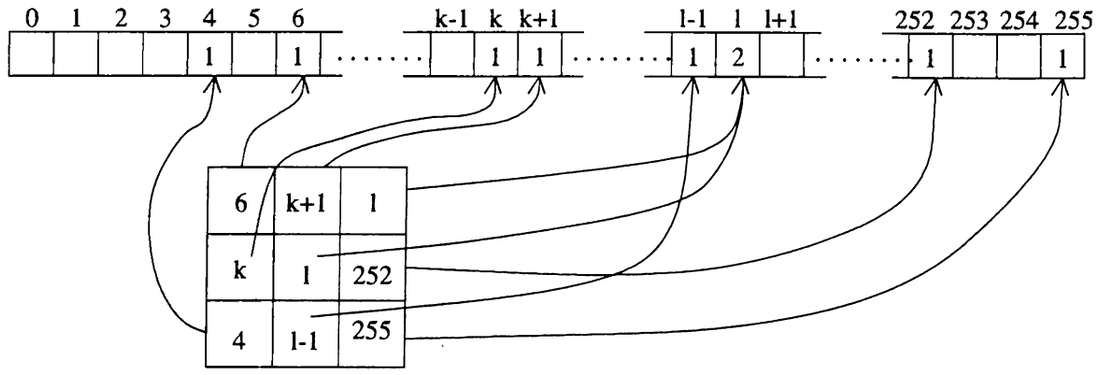


Figure 2.11: Finding the median of 9 numbers

On the other hand, the directional median needs only 3 elements, so we obtained the second large value directly by comparing the numbers.

Another process is the histogram equalization [11], [12], [13]. The equalization of the histogram in a marked window is very important especially in radiological images. Usually the dynamic range of the gray tone is poor in these kinds of images, so the digital viewing of them may cause problems. Histogram equalization is very powerful in this aspect because the details can be separated better if the image tone is almost constant in a region.

In the continuous case, one can exactly make any random variable uniformly distributed over (0,1) as

$$\nu = F_u(u) = \int_0^u p_u(u)du \tag{2.7}$$

where, $p_u(u)$ is any probability density function. In order to implement this transformation digital images, we calculate the probability of each discrete gray value and call these values $p_u(x_i)$ for the gray level x_i. If $h(x_i)$ is the number of pixels with gray value x_i, then

$$p_u(x_i) = \frac{h(x_i)}{\sum_{i=0}^{u} h(x_i)}, i = 0, 1, ..., L-1 \tag{2.8}$$

where, L is the number of gray levels. Now we can obtain another set v' with the same number of quantization level, L.

$$\nu = \sum_{x_i=0}^{u} p_u(x_i) \tag{2.9}$$

$$\nu' = Int(\frac{(\nu - \nu_{min})}{1 - \nu_{min}}(L-1) + 0.5) \tag{2.10}$$

This ν' in general will not be exactly uniform because of being a discrete variable, however its effect on images with poor gray tone range is considerably good in terms of increasing the range of gray tone. Our way of implementing the histogram equalization requires methods similar to the the implementation of the 3x3 median filtering. The past values are first accumulated in an array of 256 and the new values are extracted from that array.

Halftoning may be useful in systems having two level image display monitors [1]. Another place to use halftoning can be two level printing of the images. In the future (when a printer is added to the system), the user might want to print a marked part of the image. In this situation, halftoning (possibly without displaying on the screen) is an essential tool.

The implemented "halftone()" function works in a similar way to the "smooth()" function. We supported two types of halftoning. One applies a comparison with the Bayer Ordered Dither Matrix [1], given as a 2-D array, and the other simply adds a uniform distributed random number to the image pixel values and quantizes to zero or one. Both of the methods are implemented in a single function and the method is choosed by an indicator variable.

The Bayer ordered dither matrix is :

$$\mathbf{Bayer}_{4x4} = \begin{bmatrix} 15 & 143 & 47 & 175 \\ 207 & 79 & 239 & 111 \\ 63 & 191 & 31 & 159 \\ 254 & 127 & 223 & 95 \end{bmatrix}$$

With this matrix, a constant tone 4x4 region increases its contrast in the following order :

```
0 0 0 0     1 0 0 0     1 0 0 0     1 0 1 0
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
0 0 0 0     0 0 0 0     0 0 1 0     0 0 1 0
0 0 0 0     0 0 0 0     0 0 0 0     0 0 0 0
```

```
1 0 1 0     1 0 1 0     1 0 1 0     1 0 1 0
0 0 0 0     0 1 0 0     0 1 0 0     0 1 0 1
1 0 1 0     1 0 1 0     1 0 1 0     1 0 1 0
0 0 0 0     0 0 0 0     0 0 0 1     0 0 0 1

1 0 1 0     1 1 1 0     1 1 1 0     1 1 1 1
0 1 0 1     0 1 0 1     0 1 0 1     0 1 0 1
1 0 1 0     1 0 1 0     1 0 1 1     1 0 1 1
0 1 0 1     0 1 0 1     0 1 0 1     0 1 0 1

1 1 1 1     1 1 1 1     1 1 1 1     1 1 1 1     1 1 1 1
0 1 0 1     1 1 0 1     1 1 0 1     1 1 1 1     1 1 1 1
1 1 1 1     1 1 1 1     1 1 1 1     1 1 1 1     1 1 1 1
0 1 0 1     0 1 0 1     0 1 1 1     0 1 1 1     1 1 1 1
```

According to the application, the Bayer matrix or random noise addition can be used.

2.7 Different screen utilities

There are a few screen utilities which are essential for a viewing program. The first one is the utility to save an indicated portion of a possibly modified image from the screen to the disk.

The save operation is done by writing the pixel values to the disk with the "putc(value,outfile)" command of the *C* compiler in the PPM format. The selected region is first saved to an auxiliary file. The name of the file to save as is asked to the user and then that file is renamed to have the indicated name.

The next utility is the calculation of the area inside a closed contour defined by putting points with the mouse (figure 2.12 shows how this is done graphically) [15]. This is possibly a useful utility for the radiological users. The application may arise when the same tissue of the same person can be examined at different times. The doctor may want to see if there is a change in the size of a region. After marking the surroundings of a specified part, the following function calculates the inner area by approximating the green's

integral for the discrete case. In between the sequential points, we assume a line connecting them, and the last point is assumed to be connected to the first point in the index with a line. The unit of area is the number of pixels in the original scale. If the Image is magnified, the function that calls the find_area routine takes care of the scaling, so there is no confusion in terms of the area. Unfortunately, the user must consider the scaling during the scanning operation. If the two images are scanned in two different scales, there is no reason to compare the two areas in those images.

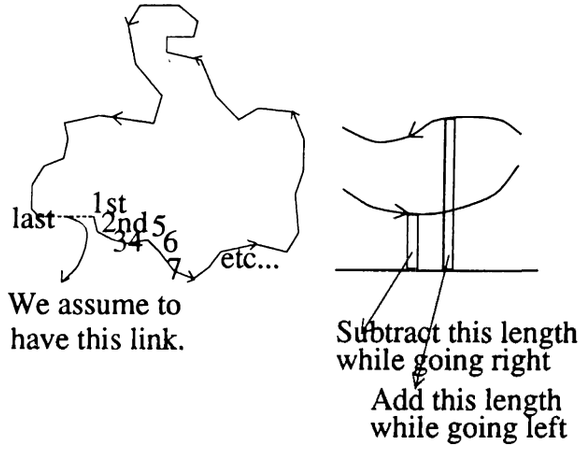


Figure 2.12: Finding the area

Another utility about the user interface point of view is being able to write text on the graphics screen. If the user wants to put some labels on some places, he or she can press the "T" key at any mouse position and write anything by using the keyboard. When an "enter" is pressed, the text mode finishes and any pressed key becomes the operation, like "Z = zoom", "S = L.P.F." in the normal mode. After drawing an arc shaped arrow by putting points, the label can be typed to indicate the contents of the place the arrow is positioning. After that, these changes can be saved by using the "save_screen()" function which is called after drawing a rectangle to indicate the borders and pressing "D". This enables us to store the modified images. These modifications include any kind of filtering, histogram equalization, halftoning, point drawing and text inserting. Color map modification is not a direct modification on the image data, so that information cannot be saved.

Now we will briefly describe the WriteText() function which outputs a text on the graphics display. It uses the ROM BIOS data to find out the shapes of the ASCII characters and then puts points on the filled positions for each

letter. In order to get the font shapes present in the ROM BIOS, we do the following :
AX = 1130h
BX = 0600h
call INT(10h)
segment of "font" = ES
offset of "font" = BP
so now there is a pointer at the address "font" to the shapes of the data. In order to find if a pixel position of an ASCII character "ch" is filled or not, we obtain another byte "test" as
test = *(font + ch * 16 + j)
where for values of "j" running from 0 to 15, the eight bits of the "test" variable corresponds to the filled bit positions along the row of the jth column. This shows that every visible ASCII character is 16 bits long and 8 bits wide. From now on, we can easily compare the bits of this "test" variable with 1 and 0, and put appropriate pixel values on the screen corresponding to the bits with value 1.

We implemented two different write modes. In the first mode, we make put-point operation on the filled bit places with pixel values 100 more than the background pixel values, and in the other mode, 100 less than the background pixel values. In this way the <backspace> clearing is enabled because when that key is pressed, we can go to the previous character position and put the same character in reverse mode (100 less if the previous one is written by putting 100 more). This method recovers the background exactly.

This text writing utility enables another feature too. In this way, we had the ability to communicate with the program without leaving the graphics mode and then re-drawing the whole image. As an example, when the area of the closed contour is found, it is written directly on the graph and then removed. Yet as another example, the file name that is asked after saving the portion of the screen is written directly on the graphics screen.

2.8 The main control function

The control of all these drawing and other utility functions are shared by the main function and the mouse handling function. The mouse control function is called by the main function in a loop manner, and the filtering and other window operations are called within this mouse control function. When we need to redraw the image such as in scrolling or zooming, the mouse controlling function passes the control of the program to the main function by setting some flags to indicate the last position of the mouse arrow and the drawn rectangle.

The present protocol between the control functions can be described by the following flow chart.

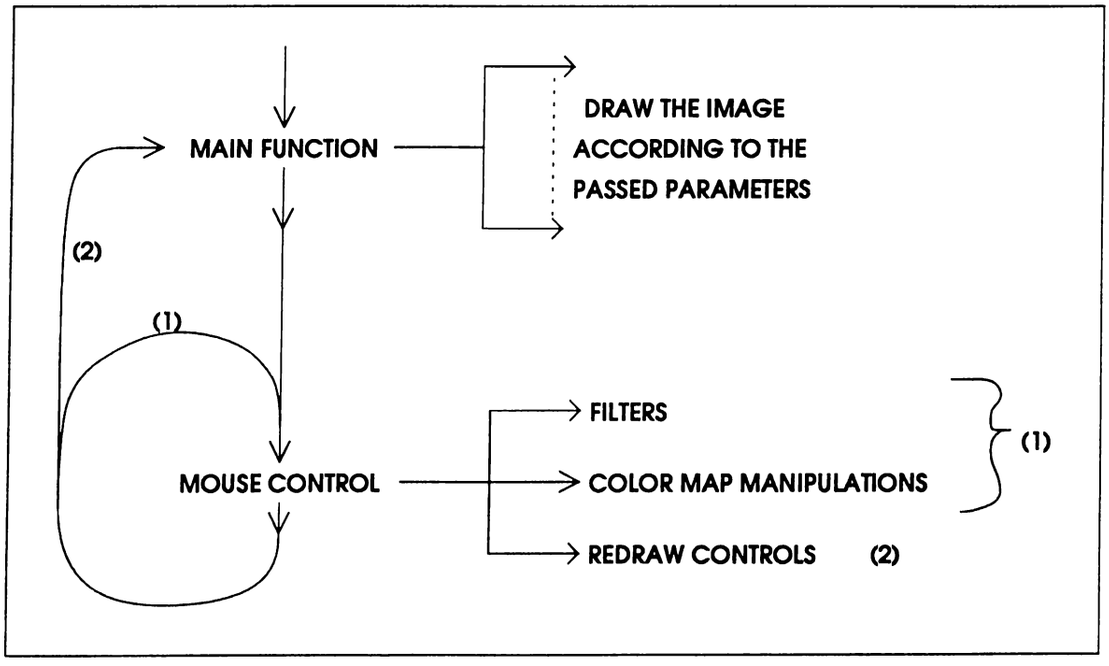


Figure 2.13: Program control flow chart

The mouse control function (fig. 2.14 and 2.15), takes care of all implemented operations either by directly executing the operation or by quitting after setting some flags. Each operation that require quitting sets different flags. As an example, the image shifting operations set the horizontal or vertical offsets and leave the control to the main function, but the zooming operation activates the flags about the corners of the window. These parameters are then controlled by the main function and the new drawing modes are decided.

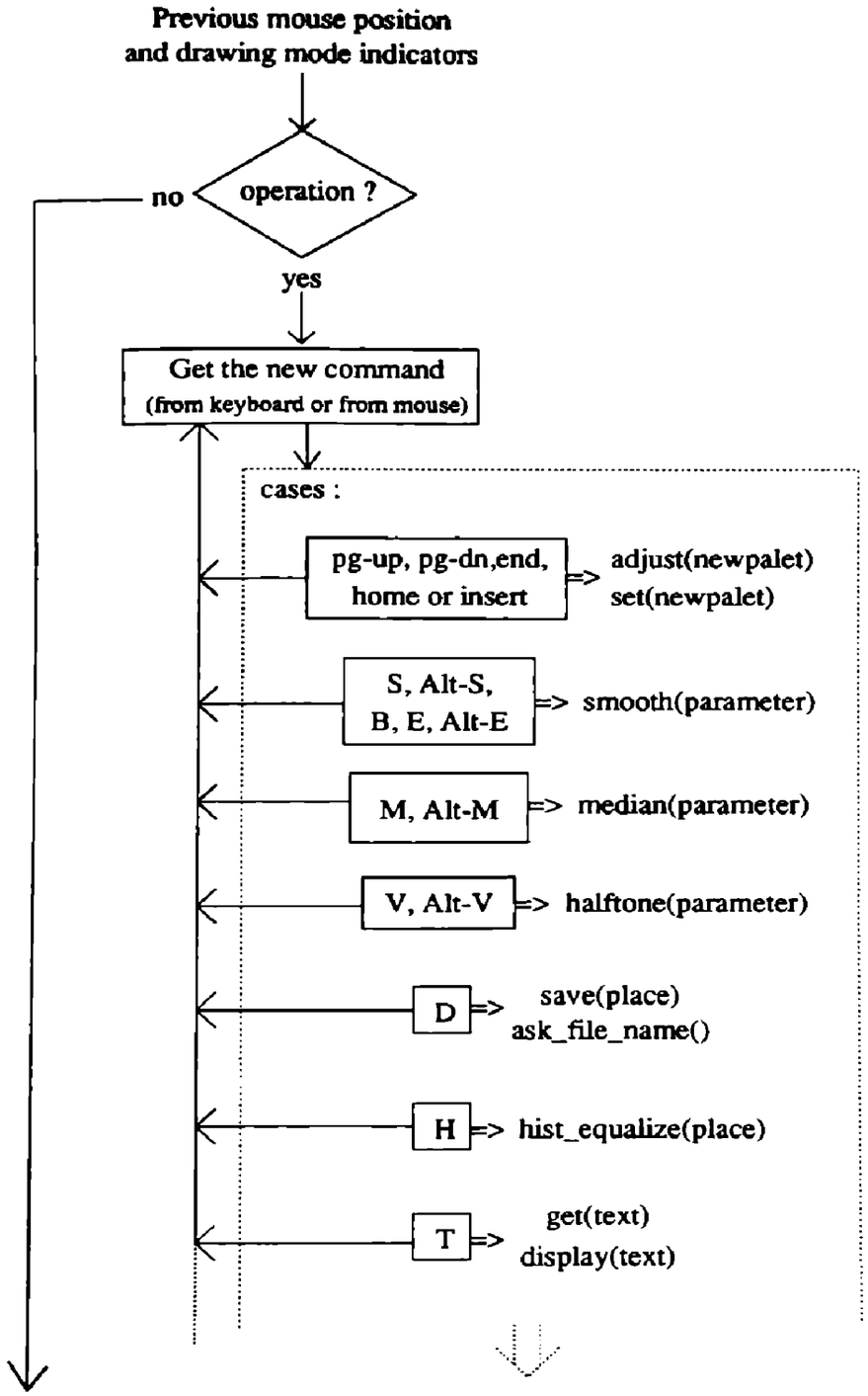


Figure 2.14: Mouse control function, continue...

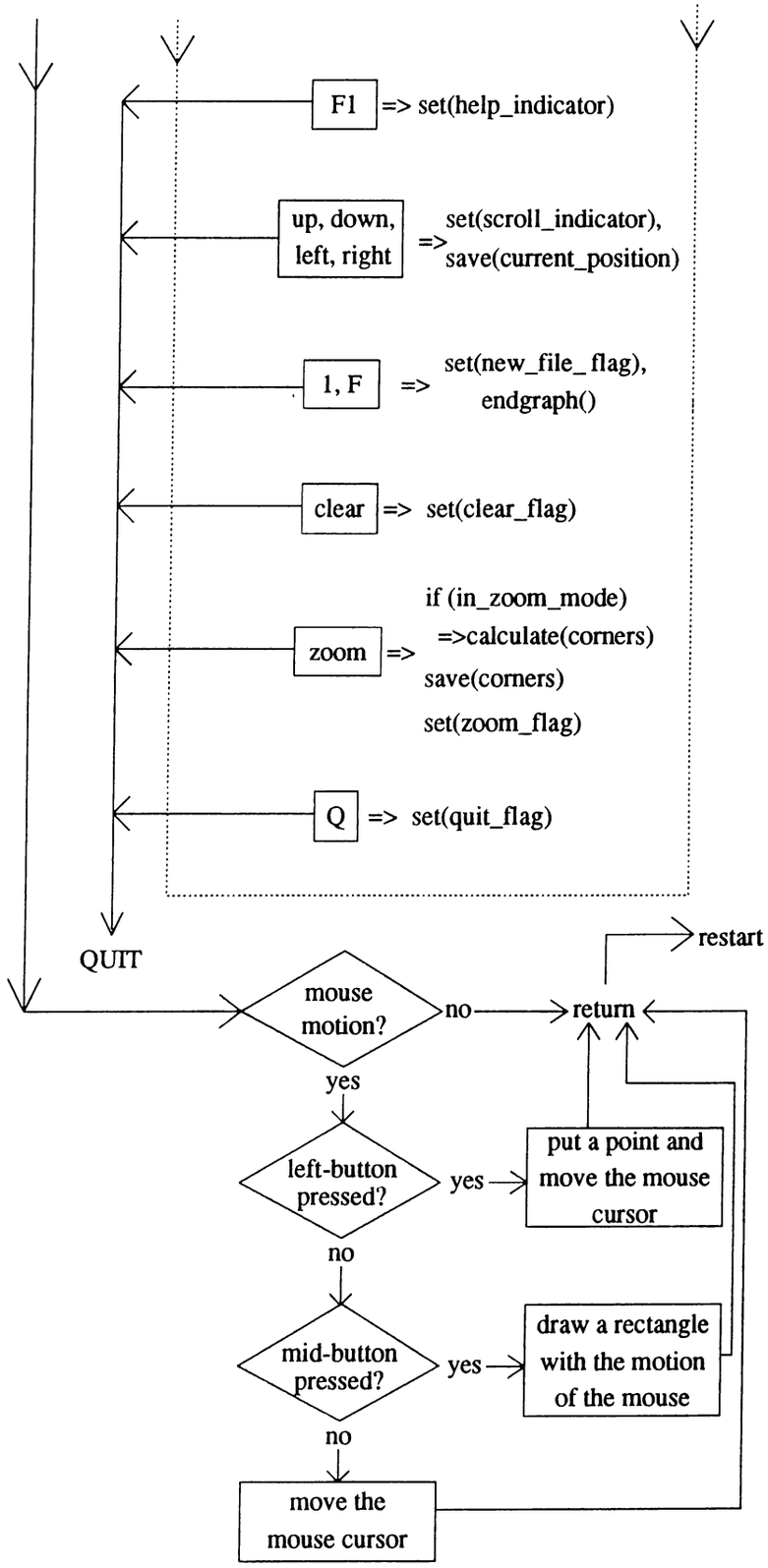


Figure 2.15: ... continued

As explained in the above lines, the mouse control is executed in a loop within the main function as long as a quit or a re-draw operation is performed. Figures 2.16 and 2.17 describe the flow of the main function and its relation with the mouse controlling function.

Other than the implemented operations and utilities, many other operations can be integrated in this program. Nevertheless, some number of operations are intentionally excluded in the software. As an example, we skipped geometric processes such as rotation and advanced image enhancement techniques such as deconvolution in this software. The reason for excluding geometric processes is basically its uselessness. Since the user has the scanner, he or she can easily scan the image in any orientation he/she wants. Another reason is that it requires PutPixel operations to draw the rotated images because the frame buffer scans the screen in a top to bottom manner.

We skipped the implementation of operations like de-blurring or edge enhancement because there are two big constraints about these kinds of operations; memory and speed. Since we do not have the image in the memory, each time a pixel value is needed, it would either be requested from the screen or from the disk. We tested a 64 by 64 image filtering by using a filter of size 16x16 and it took 6.5 minutes if the pixels are obtained from disk and 4 minutes if they are obtained from the screen by the getpt() function. Although the size 64 by 64 is enough to handle by the semiconductor memory, we used the above methods to be comparable with the 800 by 600 case which is the screen size. The size 16x16 is selected for the filter such that the windowed version of the inverse filter performs reasonably well, that is, an intentionally blurred image gets recovered visually. The result of the test shows that, the time required for these kinds of operations on large images is unacceptable for a practical user.

Supporting many more different image file formats is also a utility that can be considered, however this subject is not very important for our application. The scanner and the viewer are compatible in terms of file formats because both of them support the TIFF format. As a result, the user can use the workstation without encountering viewing problems. By making use of the JPEG format, we also obtained a possibility to compress the image files. In order to be compatible with the JPEG program, two other image formats

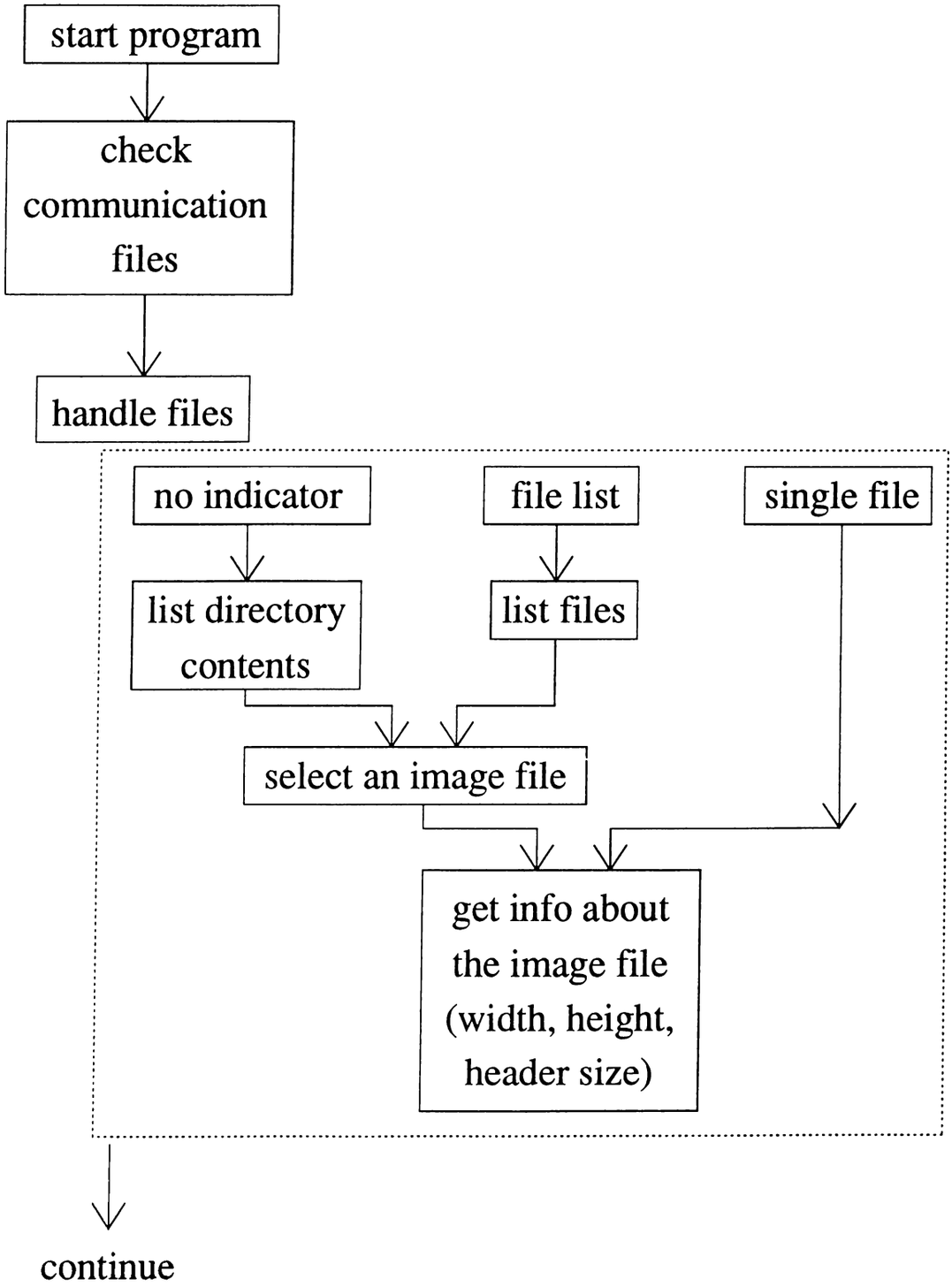


Figure 2.16: Main function, continue...

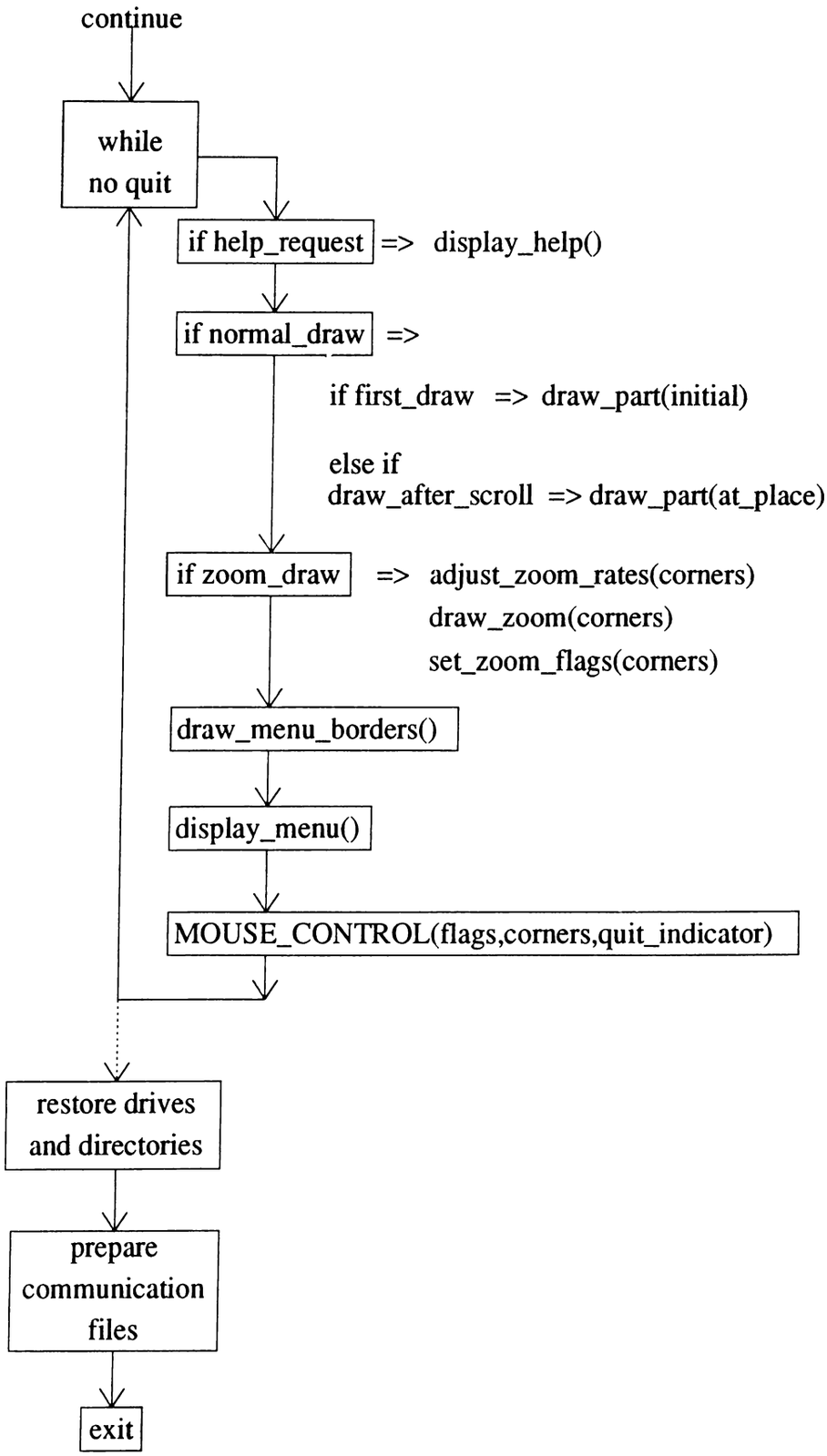


Figure 2.17: ... continued

(Targa and PPM) are supported and the conversion between the necessary formats are enabled.

The rest of the file formats would enlarge the software and consume more memory, which is very important and critical for this program. The brief descriptions of the two file formats JPEG and TIFF will be given in appendixes A and B.

The next chapter will concern about the data base handling program.

Chapter 3

THE DATA BASE SOFTWARE

In this chapter, the items of the data-base program will be interpreted. The name of this program is "MIW.EXE" which stands for the "Medical Image Workstation" and it is the root of all other imaging programs. By saying *root*, we mean that this is the program that is executed at the beginning and other programs are called from this program so that their existence is occluded. The user should be unaware of the other independent programs.

3.1 Record structures

The data-base program uses a patient record to keep the necessary data. The items that form the patient record fields are

```
typedef struct {
  char name[30];
  unsigned int age;
  unsigned int id;
  char address[100];
  char notes[70];
  char filenames[30];
} patient_type ;
```

This record has a total size of 234 bytes for the PC. A data file of 2000 records must be present for the program to work properly. This data record is used for all patient records and its structure does not change for special uses like different patience types.

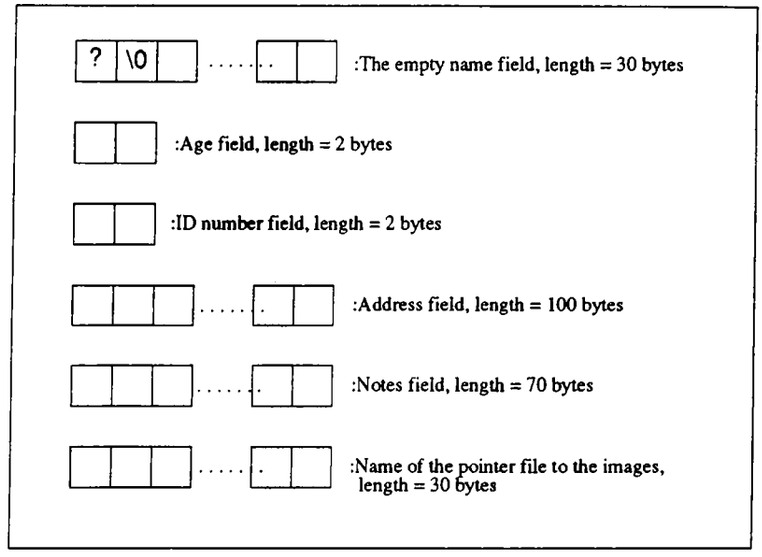


Figure 3.1: Structure of an empty record

The data base management [16], [17] is done by using hash indexing and a considerable amount of free space for a low load factor (used space / available space).

Initially, if this record file is not present, the user can activate the "Create a record file" option at the start menu and call the function "creat_record()". The flow of this function is as follows:

1-) Display warning about file creation.
2-) Get the name of the file to be created.
3-) Assign a file pointer to that name.
4-) Initialize the patient record fields
4.1-) record.name = "?"
4.2-) record.other_items = 0
5-) For 2000 times:
5.1-) Write the record to file

5.2-) If write fails, warn the user and quit
6-) Acknowledge the completion of the operation.

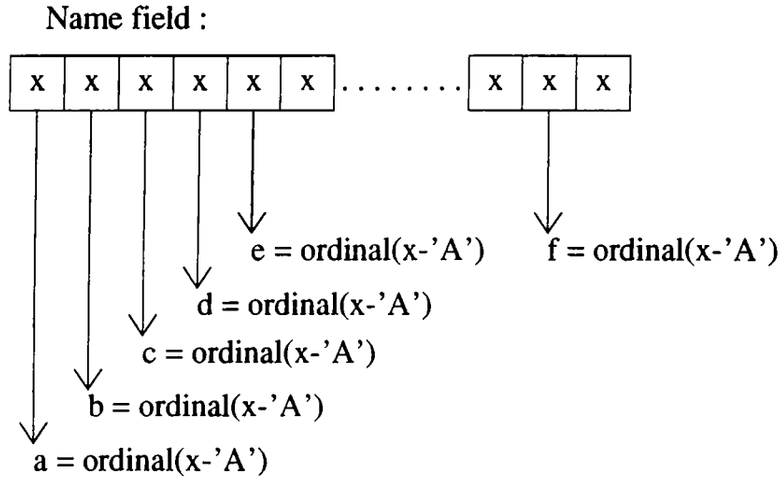
An empty record is indicated by a "?" at the name field, so when an insertion is necessary, the program searches for a place with the name field equal to "?" or "*". The significance of "*" will become clear when the record deleting operation is introduced.

The reason for having a pre-built record file is the method of searching, namely "HASH" citedatabase. The record file must be ready with the fixed size for hash indexing. We know that hashing is very fast and efficient if the index generating keywords are determined well. There are two more methods that are commonly used in organizing files. These two methods are considered in the group of "Indexed Sequential Organization". The first of them is the "indexed sequential access method" (ISAM) and the next is the "virtual sequential access method" (VSAM) [16]. These methods are for the case of inserting and deleting blocks of data in the disk space by indexing the positions of the records. The difference between the two is only in terms of the indexing methods. In ISAM, both random and sequential access to the records are possible, but the indexes of the present records must be rearranged after each insertion and deletion. In VSAM, in addition to these utilities, there is no need for index rearranging and the file can change size arbitrarily as a result of its indexing method. Details about these methods can be found in page 100 of [16] and page 40 of [17].

As opposed to the previous methods, hashing has an advantage of rapid access to individual records and simplicity of implementation. However the sequential access to record items is difficult in hashing. As a result of this, we did not implement the option of listing all the patient records in the file. Another draw-back of hashing is the inefficient use of disk space in the case of low load factor.

A good hashing routine should distribute the records as evenly as possible over the available address space [17]. The name field is used as a key for indexing in this program. The aim is to minimize collision of the index numbers and this can be done by logically arranging the function by increasing the function complexity. We tested 150 regular Turkish or English names about collision problems. Occasionally, only two collisions occurred. Nevertheless, the

program is designed to handle the collision cases pretty well. A more severe problem is for the people with exactly the same name, but we also proposed a solution for this case. The solution is that we skip to the next same named person after the displayed one upon request.



num = 20*(a) + 10*(b) + 5*(c) + 2*(d) + (e) + (f)

HASH index = (num) mod 2000

Figure 3.2: An illustration of the HASH index calculation

This kind of an indexing approach is suitable for our case. In the average where the record file is moderately full, we know that the average time required to access a record is approximately equal to the time for accessing a record in the non-collision case divided by a constant [16]. There is another good point this method. If the file becomes nearly full, the order of the search time does not change, instead of that the scaling constant possibly changes. Since this constant is a function of the length of the word to calculate the hash index, even in the worst case, the search time does not exceed a certain value.

We do not have much possibility to test the full file behavior of our indexing function. One can guess that it takes a lot of time to type 2000 reasonable names on the keyboard. If the file reaches a certain load ratio, the user should generate a new record file and store the new records in that file.

3.2 File structures

The three basic functions for the record manipulations are "record insertion" , "record deleting" and "record access" [17]. Although very well established rules exist for these functions, it is not easy to implement them. Especially the behavior in the case of collision is hard to handle.

The reason that the pointer goes to the k-th place for deletion in figure 3.3 is that usually more than 64Kbytes far from the start point ((234x2000) > 65536). By incremental seeking, we ensure that the operations are done within an integer range.

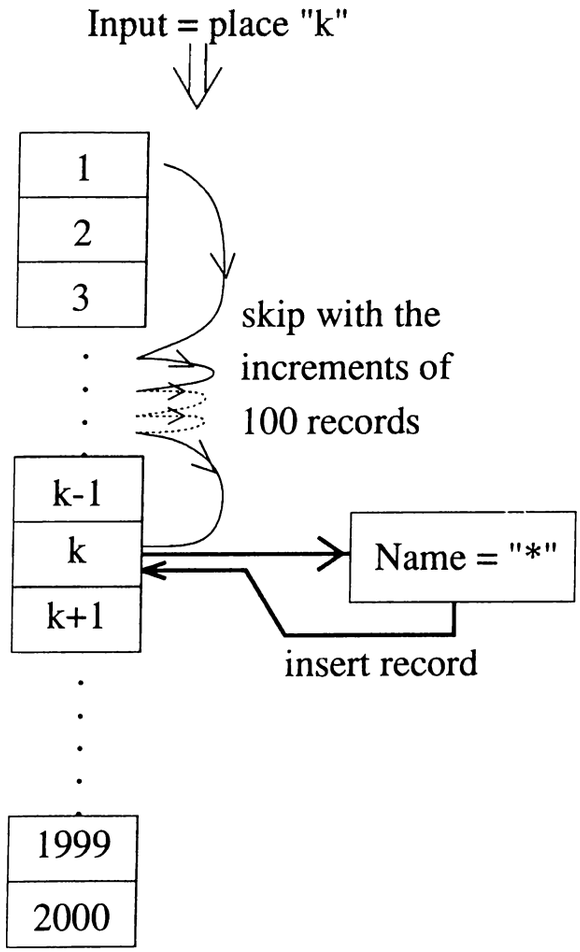


Figure 3.3: Deleting a record

We put a "*" in the name field instead of a "?" for the deleting case. This is because there might exist some other records with the same index number after this deleted record. If we put a "?" mark there, the search function stops at

the place of this question mark and the rest of the records with the same hash index would be lost. With the implemented method, the search function goes on searching after the "*" mark, furthermore, the insertion function considers that place as empty and can put a record there if the index matches.

The insertion function works as described in figure 3.4. It also handles the modification business in the case of updates. If there is an update, the "index_position" takes the value corresponding to the record to be changed, otherwise, it takes the value -1 indicating that the incoming data belongs to a totally new record.

```
function : Put_Item
inputs : name, index_position
output : a_record


if (index_position = -1)
    place = hash(name)
else
    place = index_position
```

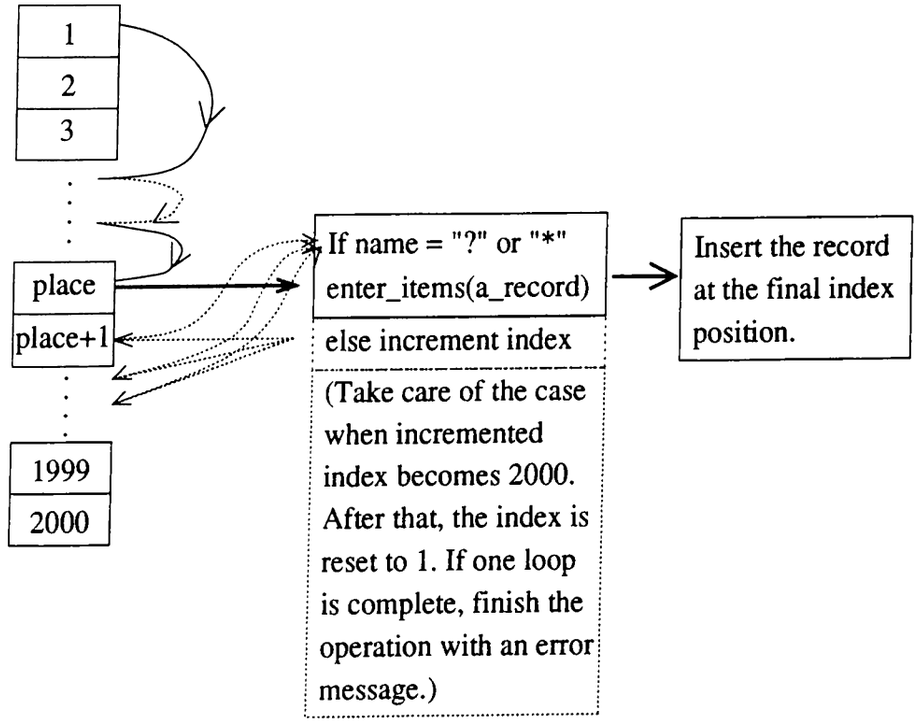


Figure 3.4: The insertion function : Put_Item

The following figure describes the flow of the record access function.

```
function : Get_Item
inputs : name, index_position
output : a_record


if (index_position = -1)
    place = hash(name)
else
    place = index_position
```

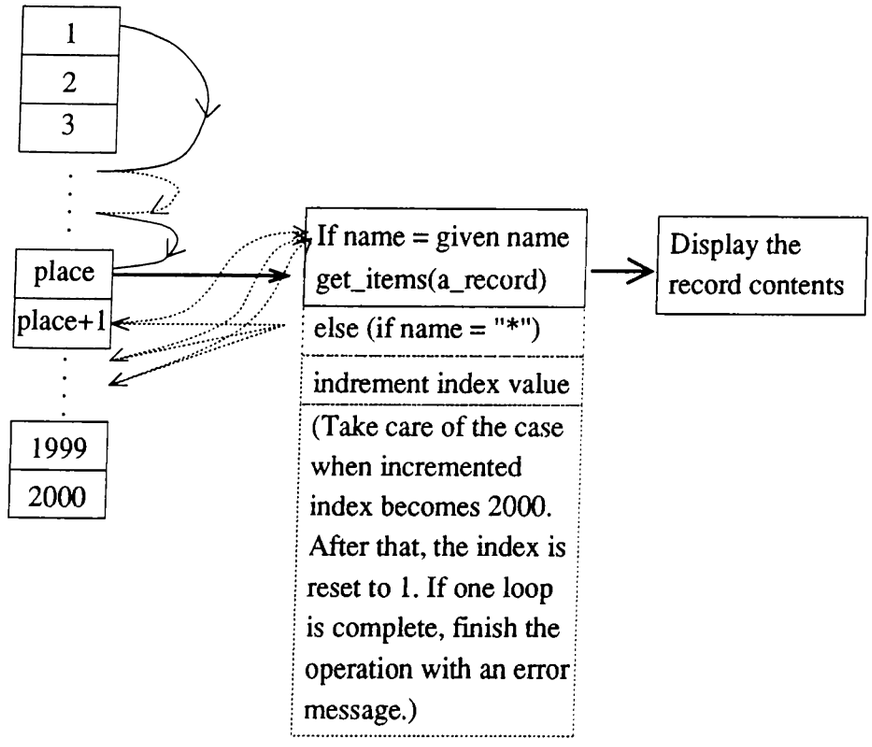


Figure 3.5: The access function : Get_Item

The "index_position" variable is used in order that there might be a need for searching another record with exactly the same value in the name field. This operation will be clearer when the record access function is described.

These two functions satisfy our constraints about record manipulations, so we use them as they are. However, one can notice that lots of control statements are necessary in the functions, but they are there in order to avoid any kind of bugs. If the application of this program is considered, it is seen that the program is not bug tolerated.

Actually, the record access function is not as complicated as the insertion function because the place to go is obvious from the hash function and the rest is just comparison of the names. What is tricky here is the option to start searching from the place "index_position". If that variable is a number other than -1, the search starts from that number. This option is present for the situation to handle the "same name" ambiguity. In the program, when a person record is searched, the first entry that matches the name comes to the screen. If this is not the person that the user wants to investigate, the "Get_Item" function is called repeatedly again by setting the "index_position" as the new place. As a result of this utility, the search does not stick to the first entrance.

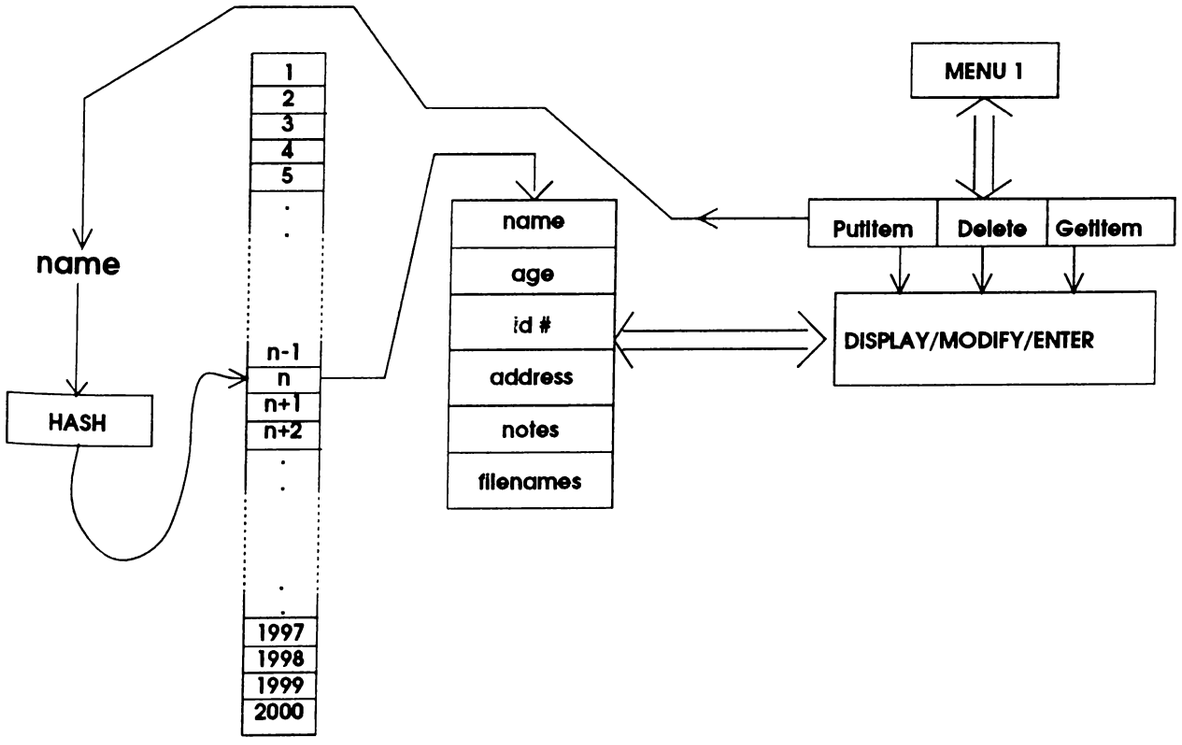


Figure 3.6: Operation of Put_Item, Get_Item and Delete

The incremental seeks in Put and Get_Item are because of the same reason as in the "Delete" function case.

3.3 The flow of the program

The implemented methods are suitable for our applications in a small environment like the single PC. There might be some other methods which are probably more efficient in terms of data and file handling, but to obtain better codes for a data base handling program is again a software engineers job.

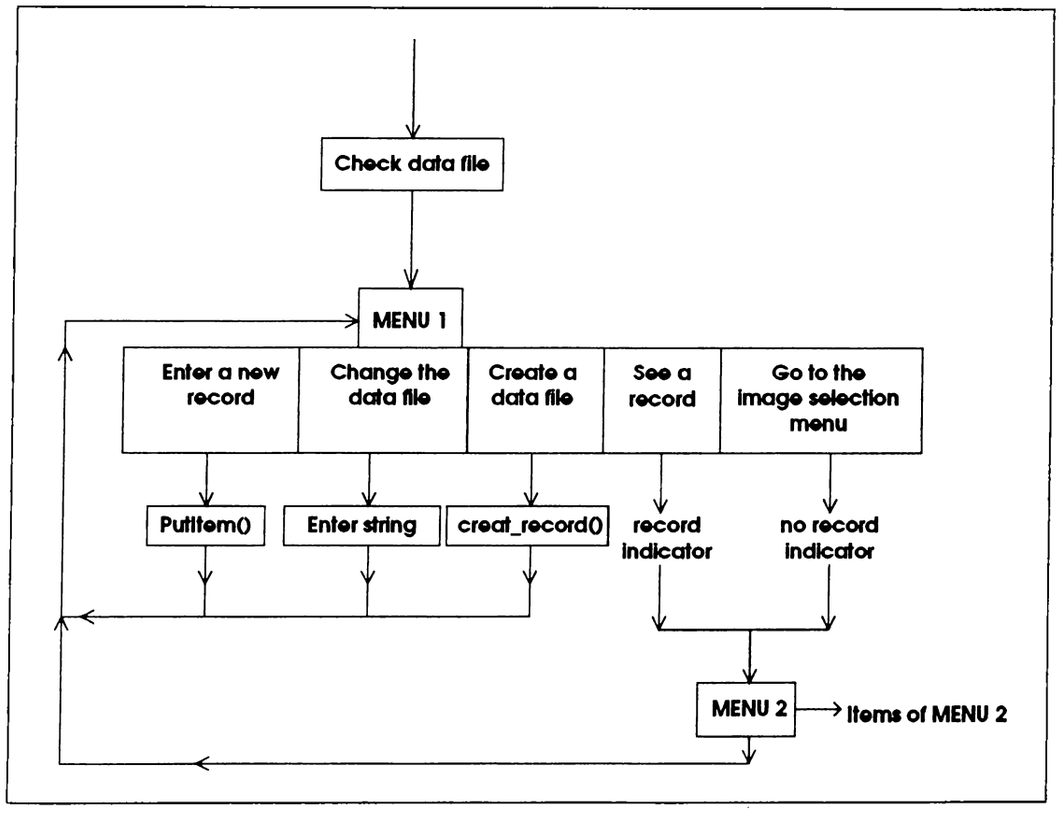


Figure 3.7: Main flow diagram

A topic which is assigned to be relatively less important is the user interface part. A good interface could be obtained by utilizing some pop-up or pull-down menu styles, but this is beyond the scope of this thesis. Unfortunately, it is usually the user interface part which sells a program or a system . Because of this, we tried not to make the program a totally non usable one. We guess the menu system described below is enough for a minimal user interface.

The menu system is composed of two hierarchical components, "menu1()" and "menu2()". The first one is called by the main function and it displays the main menu which are composed of :

1) See a record
2) Enter a new record
3) Go to the image selection menu
4) Change the data file name [TheFile]
5) Create a data file
Q) Exit program

This menu items are activated either by pressing the appropriate number (1,2,..,Q) or by pressing the mouse button when the cursor is on the corresponding line.

When the first item is selected, the name of the person to search is asked, its index is found by the hash function and finally the second menu is called by passing this index as a parameter. The second menu displays the contents of the record field and asks for the next operation. A more detailed description will be given in the next chapter.

If the second item is selected, the situation is simpler and the function only calls "Put_Item(&patient,0,0)" and checks its result.

The third item also activates the second menu with the index parameter -1. In fact it is the second menu function that executes the viewing program. When the index is -1, it easily understands that there is no record insertion, so it directly executes the viewer by writing some parameters in a dummy file whose name is known by the viewer.

The fourth item does exactly what its name requires. It asks for a new string as the data file name and writes it to the configuration file. This configuration file may or may not be present before the execution of the program. If it is not present, the program assumes the default record data file name "dat.fil", otherwise it takes the name of the data file from the contents of the configuration file.

The final item before the exiting option is to create a new and empty data file. When this item is selected, the "creat_record" function is called and a data file is created with the given file name.

The second menu is somewhat more tricky and more problems are encountered during developing the function for the second menu. This menu displays

the contents of a record and asks for the next operation. Its display structure is as follows :

press 'i' to select the image files or

press "ESC" to search directory or

press 'd' to delete the record from the file or press 'u' to update the record contents or

press 'g' to go on searching another "patient.name"

press 'q' to return to the main menu

Name :patient.name

Image file :patient.filenames

ID number :patient.id

Address :patient.address

Notes :patient.notes

Age :patient.age

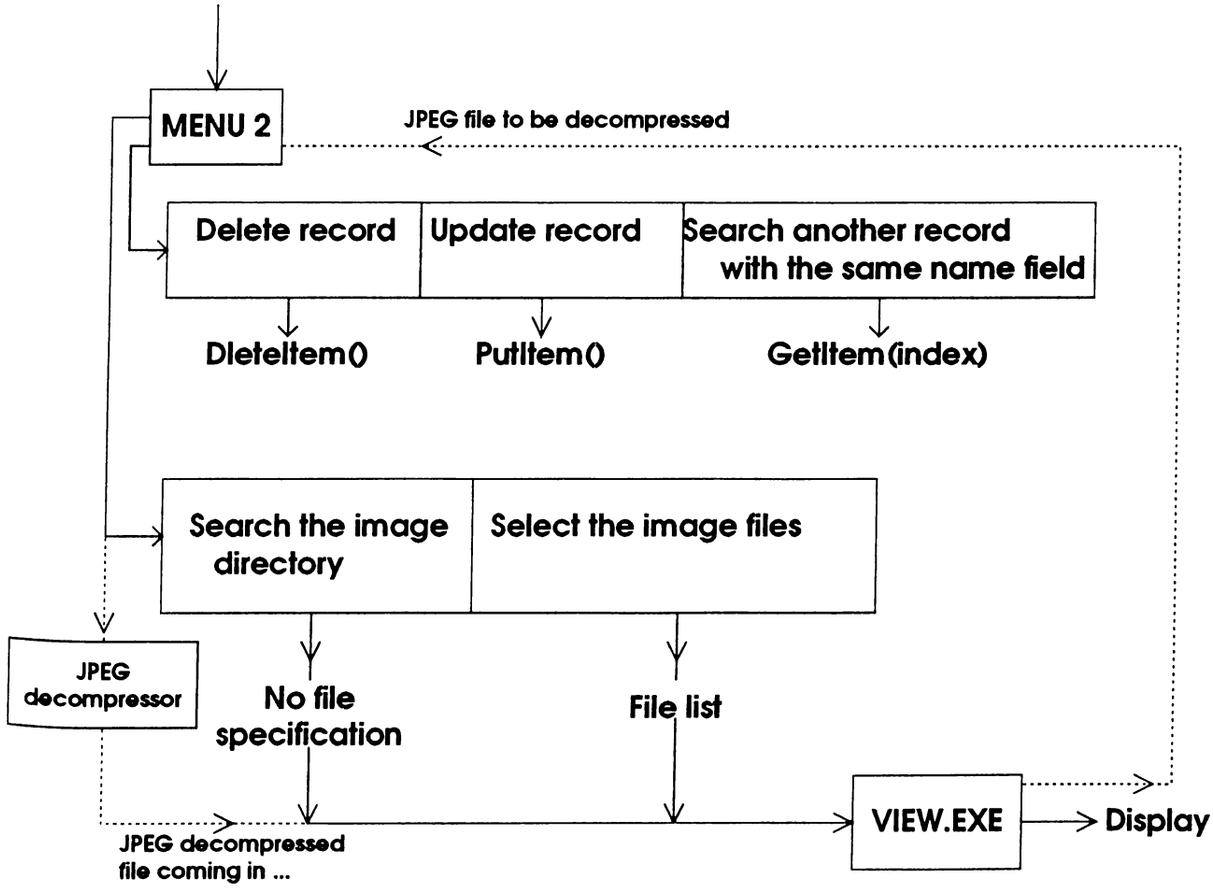


Figure 3.8: Second menu and viewer program interconnections

Similar to the previous menu, the user can press the appropriate key or press the mouse button on the appropriate line to activate an item. Notice that when the index is passed as -1 from the first menu, the viewer is called directly without displaying this menu.

The 'd' choice activates the "delete(index)" function and checks its result. For the 'u' choice, the "putitem(&patient,1,index)" is called.

The case of the same name ambiguity is handled by the 'g' item. By calling "Get_Item(patient.name,&patient,&index)", the search is started from the place one after the present index as described before.

The relatively problematic parts are the "ESC" or 'i' options because they result with executing the viewer program "VIEW.EXE". For execution of a program, the "spawn" function of the C compiler is used. The usage is as follows:
spawnl(P_WAIT,"c:\path\view.exe","c:\path\view.exe",NULL)
In this way, the root program waits until the termination of "view.exe".

The problems about this operation comes from the memory limit. The use of "spawn" needed enough memory place to run both programs at the same time.

The next problem about these items is the need of extensive controls about the parameters passed to and received from the viewer program. A typical case is the JPEG decompression request. If this request is received by the second menu after executing the viewer, the JPEG decompresser should be executed with the received input image file name and the viewer should then be re-executed after passing the name of the resultant decompressed image file. Here, the JPEG de-compressor is executed by using the same "spawn" function. The reason for the JPEG decompression to be called from the data base function instead of directly from the viewer function is that three programs fail to load to the memory and run at the same time. It is experimentally observed that when JPEG de-compressor is executed from the viewer which is called from the data base program, the whole thing blows up and the system halts.

The need for controlling some special situations arise in the case of passing the image file list of a patient to the viewer too. In this case, the viewer

must be informed that the file list consists of only the indicated file names. Furthermore, if the image viewer is called directly by pressing "ESC", the program must support an option in the viewer program such that a file in the images directory can be added to the image file list of the current patient record. Conversely, this adding utility must be avoided if the viewer program is not called from the second menu.

Due to the large amount of code to handle these control situations, we do not present the source of these two menus here.

The integration of the operations are done in the menu1() function. The main function calls menu1() and for record investigation purposes, menu1() calls menu2(). Unless the user wants to examine another record, the program remains in menu2() function from where all the viewing and JPEG compression - decompression stuff are executed. The used programs must be ready at their required directory in order to avoid confusion in the execution protocol.

Chapter 4

EXPERIMENTAL RESULTS

4.1 Experimental results

Our system with the connected WORM optical disk and the scanner required 1MB of conventional and 2MB of extended memory. What is more, for the save and JPEG compress - decompress purposes, about 5MB of free disk space is required.

Having these requirements satisfied, together with a math co-processor and a 33MHz clock speed, we obtained the following results.

For the case of full sized window operations (800x600),

Low-pass filtering (all kinds) :	15 seconds
High-pass filtering (all kinds) :	15 seconds
Median filtering (3x3) :	20 seconds
Median filtering (3x1 + 1x3 directive) :	11 seconds
Histogram equalization :	16 seconds
1-bit quantization :	11 seconds
Saving :	11 seconds
Modifying the color-map :	~0.1 seconds

The full screen window is actually not the usual way to utilize these functions. If the user applies these operations on a smaller window such as 200x150, usually the time required does not exceed two seconds. It is noticeable that this much time is acceptable for a radiologist or any other user.

4.1.1 Fast image drawing

The difference of the time required between directional and 3x3 median filtering shows that the bottleneck in these functions are the PutPoint and the GetPoint operations which execute the interrupt 10h. Obtaining the result of this interrupt takes considerably more time than other memory transfers and arithmetical operations. This gives us a hint about increasing the speed in the case of drawing.

An obviously fast operation is the data transfer from one memory location to another place, specifically the frame buffer. Even after the control statements and arithmetical operations, dumping a block of image data to the frame buffer improves the speed by more than an order of magnitude. The control statements ore for the switching between the screen portions which have the same frame buffer address as we discussed in the second chapter.

This kind of dumping stuff also enables us to use the block reading function fread() more easily. This file reading function is much faster than sequentially using fgetc() which gets one byte of data from disk.

While drawing the image on the screen, the user easily notices that the parts of the image come to the screen in a discrete manner. This is because of reading a data block corresponding to 20 lines of the image, and then dumping the data to the frame buffer. One should take care of the place of the frame buffer pointer. It must point to the correct portion, and the pointer index must be incremented correctly to go to the next line of the image. The reason for these controls is that the image width need not be exactly 800.

By using the draw_part() function described in the second chapter, an image which with size larger than the total screen size completely appears on the screen in about 2.5 seconds. This amount of time is considerably less than any of the previously described operations. Because of this fact, a similar direct memory transfer method is used in the zoom drawing function. After calculating the interpolated points of the scaled image, the appropriate pixel values on the corresponding place of the frame buffer are changed. This way of drawing is again twice faster than directly putting the calculated pixels values on the screen just after their calculation. Unfortunately, the direct use of the

frame buffer increases the complexity of the code for the previously described filtering like operations. In the case of performing the convolution

$$v(m,n) = \sum \sum_{(k,l)\in W} a(k,l)y(m-k,n-l) \tag{4.1}$$

nine GetPoint operations are applied for the calculation of a single pixel value. The similar GetPoint operations are also needed for a 3x3 median filtering, however for the directional median case, we need 3 GetPoint operations in one direction and 3 GetPoint operations in the other direction, total of which makes 6 interrupts for a single pixel. In addition to this, sorting 9 integers takes longer than sorting 3 integers twice. As a result, the 3x3 median filtering is considerably slower than the directional median filtering..

It is well known that the scaling operation actually needs low pass filtering after zero padding.

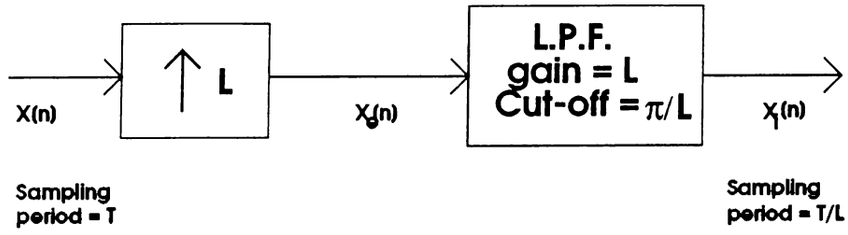


Figure 4.1: General system for interpolation

In the frequency domain and for the one-dimensional case, this corresponds to

$$X_e(e^{j\omega}) = \sum_{n=-\infty}^{\infty} (\sum_{k=-\infty}^{\infty} x[k]\delta[n-kL])e^{-j\omega n} = \sum_{k=-\infty}^{\infty} x[k]e^{-j\omega Lk} = X(e^{-j\omega L}) \tag{4.2}$$

In the time domain, the filtered signal becomes

$$x_i[n] = \sum_{k=-\infty}^{\infty} x[k]\frac{sin[\pi(n-kL)/L]}{\pi(n-kL)L} \tag{4.3}$$

In the non-integer zoom rate case, the "practical" implementation is done by inserting different number of zeros between pixels. For example for the zoom rate of 1.5, 1 zero for one pixel and 2 zeros for the other pixel are inserted

repeatedly. Ideally, similar things should be done for our zoom purposes, but usually the size of the convolution filter for low-pass filtering the interpolated signal is not less than 10 by 10. It is difficult to handle a convolution of this size since even 3 by 3 filtering takes pretty long time. Furthermore, the memory requirements become unacceptable for big filter sizes.

As a result of these restrictions, a practical and an acceptable solution is the first order interpolation. Experimentally, for reasonable amount of zoom rates (up to three times), the first order interpolation is satisfactory for visual purposes. The amount of time required to put a zoomed image on the screen is 4.5 seconds with this method.

The time for saving the edited portion of the image is actually determined by the speed of the used disk. For a good performance, both the sequential r/w and the random access of the disk must be sufficiently fast. There is not much to do for increasing the save time of the program. Yet, the saving operation takes reasonable amount of time for our implemented setup.

4.2 Obtaining the filter coefficients

As described in the second chapter, the FIR2 command of MATLAB is used for finding the filter coefficients used in the program. There is an important point to notice here. Actually, there is not much to do for the FIR2 function in the 3x3 case. A window of this size is so small that the FIR2 function fails to find a 3x3 filter that satisfy our constraints. Because of that, we relaxed the constraints to find a low-pass and a high-pass filter. The other two low-pass-filters and one high-pass-filter used in the program are determined by inspection. It is a chance that they are experimentally satisfactory for visual purposes (page 468 in [11]). Another good point is that they do not produce ripple effects as a result of rotational un-symmetries. In general, the 3x3 filter coefficients in the program are not separable. Because of this, a single function is used to apply 3x3 convolution for different coefficients.

The directional median filter is implemented for supplying a good filtering for some specific noise types. After applying that filter, we saw that it is successful in some cases. If there are not thin line shapes on the image, the

directional median filter is more suitable to eliminate salt-and-pepper kind of noises because it preserves the sharpness of the corners in the images.

4.3 Installing the scanner

The scanner we installed in our configuration is the "HP ScanJet IIp". The machine with itself is an easy to use one because the operating parts are well proportioned. After installing the software supported by HP, its usage is as simple as a photo-copy machine. In fact, the scanning part resembles the photo-copy machine in terms of tracing the glass place with a beam of light.

One might think that the charge coupled monitors (CCD's) can scan the radiological images better because of the lighting conditions. In the CCD case, we could even illuminate the image from the back which would possibly be successful as we know that the radiologists use their images that way. However, a CCD scans the image of a fixed size at a fixed resolution. In terms of these aspects, the scanner of our kind is easier to use.

In order to have the scanner work properly, the PC must have the following properties :

- 1 megabyte of conventional RAM
- 4 megabytes of available disk space
- a Microsoft Windows compatible pointing device
- at least an EGA monitor
- Microsoft Windows 3.0 or higher standard or enhanced mode
- an available expansion card slot

The installation of the card necessary for interfacing with the computer is perhaps the only time consuming part of the scanner business [10], [4]. The ROM location of the card is determined by the configuration of the switch on the board. In our configuration, we had the switch positions as 1 1 1 0 which corresponds to the ROM address C8000 - CBFFF, but the position 1 0 1 0 with memory address CC000 - CFFFF also works properly with our system. Here, the other switch positions cause conflicts between the memory locations of various other applications especially the SCSI connected optical disk.

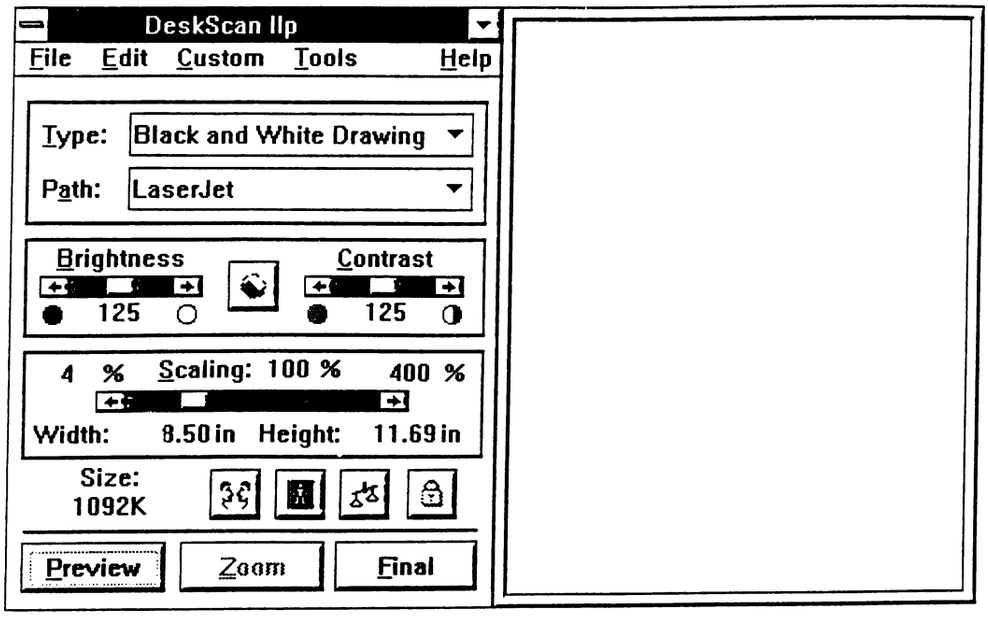


Figure 4.2: The menu of the DeskScan

After completing the installation in the proper ROM address, the user can call the application tool *DeskScan* which has a menu of the form given in figure 4.2.

This program is executed from the Microsoft Windows environment. The left side of the menu consists of the tools and the right side is the clipboard area. After putting a picture on the scanner, the scanning operation starts with clicking on the "preview" button. In this way, any image on the scanner gets scanned roughly and displayed. This operation lasts about 5 seconds. After that, by using the mouse, a region is selected from the preview display. At this position, the user can adjust the resolution by playing on the scaling menu or change the format of the image files. There are three basic modes available in this program, gray-scale photo, half-tone image, half-tone drawing. In all of these kinds, the TIFF, PCX and EPS formats are supported. After choosing a suitable format (gray-scale photo and the TIFF format for our purposes) the user clicks on the "final" menu. At this position, the program asks for a file name to save the selected area as and upon entering a file name, the scanning operation is completed within about a couple of seconds.

This much is a brief description of the experimental results obtained by using the scanner. The details about the scanner can be obtained from the document *Using the HP ScanJet IIp Scanner with Microsoft Windows* by Hewlett Packard.

4.4 Installing the WORM optical disk

Installing the Write Once Read Many (WORM)Optical Disk via a Small Computer Systems Interface (SCSI) is basically composed of three steps :
-Installing the SCSI card in the computer,
-Connecting the WORM, and
-Installing the necessary software.

There is nothing tricky in installing the card or in connecting the disk driver [10], however setting up the system and having the optical disk drive as a regular drive is a time consuming job.

The capacity of our WORM by $Maxtor^{TM}$ is 786 Mbytes per optical diskette. This is a large amount for MSDOS to handle, however after installation, the system utility files handle this situation by buffering from the hard disk. In this section, the followed installation procedure and the encountered problems will be described briefly.

After installing the SCSI card and connecting the WORM disk driver with a ready disk in, installation starts by copying the necessary .SYS files to the boot disk (hard disk for our case) and then modifying CONFIG.SYS and AUTOEXEC.BAT.

The SCSI connection of the WORM is at the BOOT part of the ROM BIOS. Because of that, the presence of the WORM connection is displayed on the screen just after the RAM-TEST of the machine. At first, the card gives an error message indicating that the SCSI address pointed by the card is not a logical address. This is basically because the optical diskette in it is not formatted. The necessary lines for the file CONFIG.SYS is as follows :
DEVICE = ASPI4DOS.SYS /P 330 /D /L
DEVICE = ASPIDISK.SYS /D

The /D switch for both files is for displaying useful information about the host adapter and attached SCSI devices during boot. The /P 330 indicates that the driver uses the port address 330 and the /L option enables the support for SCSI logical units other than zero. Actually, the port address is determined after arranging the jumper configuration on the SCSI card. We adjusted the jumper set up for drive configuration as supporting three normal drives 'A', 'B', 'C' and two SCSI drives 'D' and 'E'. The set up for address is configured to 330 because the BIOS of the PC works at port 330.

There are other advanced switches supported for ASPI4DOS.SYS but their usage was not necessary for our purposes.

Because of the /D switch, the following lines are displayed at the boot :

```
AHA-1540/1542//1640 ASPI Manager for DOS
Version 3.0
Copyright 1991 Adaptec, Inc.

     Host Adapter 0:          0      Host Adapter SCSI ID:   7
     I/O Port Address:        330    DMA Channel:            5
     Interrupt Level:         11     VDS Support Level:      MultiSegRW

     Host Adapter No:0 - SCSI ID 0 - LUN 0: MAXTOR WORM

     Int 13H active for drive E:

Int 13H routed through ASPI manager
ASPI4DOS Installation Successful

ASPI Disk Driver for DOS
Version 3.0
Copyright 1991 Adaptec, Inc.

 1 SCSI drive(s) handled by ASPIDISK
 0 Logical drive(s) installed
ASPIDISK.SYS  Installation Successful
```

The next step is to format the optical disk. In the first case, one should make a low-level format in order to make the WORM a logical drive. This pre-format is done by using CFORMAT program with the /LOWLEVEL option. After that, the utility AFDISK.EXE which is a menu driven program supported by Adaptec is used. The menus of the program are well guiding the user, so one can immediately reach the position of entering the amount of disk space. However, if the SCSI drive is controlled by DOS through the Host Adapter BIOS as in our case, this utility fails to format the optical disk. Instead of that, the FDISK and FORMAT utilities of the DOS should be used by entering the disk space value as half of the 786M. This corresponds to one side of the disk. Since there are a finite number of memory values available in the format menu, the one closest to 390M is selected.

After these operations are complete, the
0 Logical drive(s) installed
indicator becomes
1 Logical drive(s) installed
at the boot set up.

The rest of the WORM usage is no different than using an ordinary disk drive. In our system, the optical disk is installed as drive 'E', so by typing "E:" at the prompt, we switch to the optical disk drive.

The access speed of the WORM is mostly limited by the hard disk we had because the buffering of it is done via hard disk. As a result, the reading speed of a file from the optical disk is almost the same as reading speed of the hard disk. We did not use the extended memory for buffering because that kind of buffering disturbed the memory management of scanner which uses the extended memory. Cash buffering was also totally used for buffering the VGA card, so this place was not available for buffering the optical disk.

The critical point about the user is that, as its name indicates, these disks are Write-Once. If one writes something on the disk, there is no way to remove it. The DOS "del" operation still works, but it only puts a "?" at the name field in the File Allocation Table which is a Read/Write place. In the case of a delete operation, ,the places allocated by the deleted file become unusable, however, an undelete operation is always possible.

An average 700x700 radiology image (CT, MRI or X-RAY) file is of the size 500K bytes, however after JPEG compression (see appendix A) with the quality switch set to 95/100, the file size becomes 20K - 70K. After decompression, the resultant image is much better than the same image scanned with a 200x200 resolution to obtain a file size of 40K. Since the average compressed file size is about 50K, the user can store more than 10000 radiology images in a single cartridge. This number is acceptable after comparing with the amount of space necessary to store 10000 prints. Furthermore, the driver does not have a fixed disk, so the diskettes can be replaced, changed or backed up.

4.5 JPEG compression and decompression

The need for image data compression emerges from the limit of storing place. Practically, the lossless compression methods are not satisfactory for the image files. Although some radiologists are uncomfortable with the fact of lossy compression in terms of loosing the originals, computerized imaging states the need for high compression ratios. In this aspect, the transform coding techniques are very popular and well established. It is a well known fact that the Discrete Cosine Transform (DCT) [19] is the optimum de-correlating transform in the average sense for auto-regressive(1) (AR(1)) random processes. Experimentally, it is found that the real life images can be modelled by an AR(1) process with quite high success, so the DCT is a good transform technique for image data.

JPEG is a well known standard for image data compression which applies DCT (the details are given in Appendix A). The loss of precision starts at the point of transformation because even the floating point multiplications are not done with infinite precision. On the other hand, the real loss comes from the quantization operation done in JPEG. When the transformed coefficients are scaled by a corresponding factor, they are rounded to the nearest integer and the reverse manipulation (multiplication by the same factor) loses the accuracy by the amount of rounding.

It is experimentally observable that the most disturbing factor of JPEG

coding comes from the "blocking" effect where the 8x8 blocks become distinguishable after restoring the image. Actually, if the color map stays the same, the user usually cannot see this effect, but when the brightness of the image is changed by color map manipulations or when a "histogram equalization" is applied to a place, even a difference of 1/256 may get amplified and the edges of 8x8 blocks may become visible.

Similar effects may occur after some filtering operations like high-pass filtering. As a result of this, the JPEG compression somewhat lacks the capability of processing. On the other hand, if the user views the image as it is, without modification, the JPEG effects are not noticeable. Another drawback of JPEG comes from the time needed to decompress the file for viewing. For a 700x700 image, it takes about 25 seconds to decompress the file, but after decompression, the re-drawing, zooming and other operations do not require decompression again.

The user must make a choice between two facts, image quality and image size. As a result, it should be known that this Medical Image Workstation need not be used with JPEG always. It is a choice up to the user.

4.6 Data base manipulation speed

Before starting to describe the results obtained by data base handling programs, we must point out that their speed restrictions are not noticeable near those of image manipulations. A record search or image file access does not take longer than 1 second and this amount is negligible.

In the case of a full record file where lots of collisions may occur, the speed of the file position seek operation may become noticeable. We tested the following case :

1) Go to the start of record field

2) Compare the name field

3) If not the same as the given key, seek to the next record field, go to 1

with 100 collisions, i.e., the loop executed for 100 times. The operation lasted for three seconds. As a result, even the full search of the file lasts about 60 seconds. With the fact that this case almost never occurs, we do not expect a

long time for record search.

As described in the previous chapter,every patient record has a file name. In the file with that name, the image files belonging to that record are pointed. The viewing program takes the list of the image files from the contents of the previously described file. The user must take care of the consistency problem here. The image file written in the list must be present at the indicated place, otherwise the viewer and the JPEG de-compressor gets confused and nothing gets displayed after some error messages. In other words, if the file names are written incorrectly or if those files do not really exist, the program has no way to process.

A more dangerous case is when the drive of the specified file name is not ready or when the change directory operation is done to a not-ready drive such as a floppy driver with no floppy disk in it. In that case, the well known
Not ready reading drive A:
Abort, Retry, Fail?
message gets displayed. Actually there are methods to search the presence of a drive without using the DOS commands, but the DOS commands are the easiest and safest ones for proper usage. When the above message is displayed, the user must behave just like in the case of working at the DOS prompt.

A good feature of the program is that it never leaves the auxiliary files undeleted on the disk. Even when these kind of errors occur, the program clears the communication files securely, so that the user is never aware of them.

4.7 Compiling the source code

The compilation step might seem unimportant about a software, but if the code and data are both very large, the tricks about compilation get important. The efficient use of the "memory model" option [8], [9] is the key point here.

The medium memory model of the C compiler handles the situation of large code. All data must fit in one 64K segment, but code can use multiple segments. In this case, all pointers to data are 16 bits, but jumps require 32 bit addresses. The result is quick access to data, but slower code execution.

The compact memory model is just the reverse of the medium model. All code must fit in one 64K segment, but data may use multiple segments. If our software could be able to use one of the above models, it would be faster, but the size of both code and data are very large.

The bigger memory model is the large memory model. Here, both code and data may use multiple segments, but a single data item cannot exceed 64K. This is actually the memory model our software must be compiled with, but still after further tricks.

The last thing to be done is splitting the functions into subsections of 3 or 4 for the two programs. This splitting is done by collecting the splitted functions in some files and gathering those files in a project file. This kind of project method is commonly used in MSDOS [7] environments. This is essentially similar to writing "make-files" [8], [9]. The trick comes after this splitting. The compiler options must be set such that the separate modules get overlaid.In this way, the unused functions are thrown away from memory. For a fast execution of the program, one should be careful about collecting bunches of functions together in one group. If one function is used extensively in another, these two must be put in the same project item. If the items are not organized logically, the execution speed may get worse.

After a suitable overlaying, the programs and executed sub-programs work properly without disturbing the memory system.

4.8 Using the program

When "MIW.EXE" is executed, the user encounters the following menu :
1) See a record
2) Enter a new record
3) Go to the image selection menu
4) Change the data file name [TheFile]
5) Create a data file
Q) Exit program
Any item can be selected by pressing the corresponding key (1,2,..,Q) or by pressing the left mouse key when the cursor is on the same line as the item

that we want.

By selecting item "1", the program prompts :
Enter the name :
The user must enter the name of the patient correctly together with taking care of the upper-case and lower-case letters. If an empty string is entered in the name field, the program returns to the main menu immediately. If the specified name is not found, a warning message is displayed and the program returns to the main menu again.

When a patient record with the given name is found, a menu like this is displayed :
press 'i' to select the image files or
press "ESC" to search directory or
press 'd' to delete the record from the file or press 'u' to update the record contents or
press 'g' to go on searching another "patient.name"
press 'q' to return to the main menu

Name :patient.name
Image file :patient.filenames
ID number :patient.id
Address :patient.address
Notes :patient.notes
Age :patient.age
where the corresponding values for patient.* are written.

Similar to the previous case, the user can go to a menu item either by pressing the corresponding key or by pressing the left mouse button when the mouse is on the same line as the desired item.

If the 'i' item is selected, the VIEW.EXE is executed and the file list of the patient is passed to the viewer program. The viewer takes the list and displays it with the following format :

```
->image1.tif                          List:[<Enter to select,
->image2.tif                          <Space> to enter a file name,
->image3.ppm                          'Q' to return ]
->image4.tga                          Use <UP> & <DOWN> arrows
->c:\images\jpeg\image6.jpg           Press <F1> to compress file
->c:\images\private\image7.jpg        Press <F2> to delete from list
->c:\images\image8.tif
->c:\images\image9.tif
->c:\images\private\image10.tif
```

The written file names are inventory examples and the number of files in the list may be more than that will fit in one page (25 lines). The user can move on the list items either by using the arrow keys or by moving the mouse. The items scroll up or down if more than 25 files are written in the list. By pressing enter or the left mouse button, the pointed image file is selected and displayed. If $< F1>$ is pressed and if the file is not in JPEG form, that file is compressed and the name in the list gets an extension ".JPG". If <F2> is pressed, that item is excluded from the list and the list file of the patient record is updated.

A press to 'Q' in this menu or anywhere during drawing the image returns to the previous menu where the record fields of the patient are displayed.

If the "ESC" item is selected, the VIEW.EXE is executed again, but this time no image file list is passed, so the viewer displays a different menu like :

```
No image file specified!
'Q'     = return
<Enter> = list current directory [current :''the current directory'']
<F1>    = search optical disc
<F2>    = search current directory
'C'     = change directory [ current : ''current'' ]
<Space> = enter a filename
Your choice :
```

The items of this menu are activated similarly, and upon selecting a directory search item, the contents of the directory are displayed in a list similar to the case with the file list. Just like in the previous case, the user can move around

the files with the mouse, furthermore there is one more option this time :
Press <F3> to add to the list
which is not present in the previous list menu. This option enables the user to select a file name from a directory and include it to the file list of the current patient record. The rest of the items are the same, and by pressing enter or the left mouse button, the image is drawn on the screen.

If the 'd' item is selected, a warning indicator "Are you sure [y/n]" is displayed. According to the pressed key, the patient record is deleted from the record file or not.

If the 'u' item is selected, the program starts to ask the new items of the record for updating :
Enter the ID number [previousID] : enteredID
Enter the address [previous ADDRESS] : entered ADDRESS
Enter the notes [previous NOTES] : entered NOTES
Enter the age [previousAGE] : enteredAGE
Enter the name of the image files [prev. IMAGE_FILE] : enteredIMAGE_FILE
Test the existence of enteredIMAGE_FILE
If exits, warn the user and continue
Enter the image file 1 [oldfile1] : newfile1
Enter the image file 2 [oldfile2] : newfile2

.

This goes on until an <ESC> or <F1> is pressed. If the user enters an empty string, the contents of that field remains unchanged. When <ESC> is pressed at any time, the changed parts are saved, the untouched parts remain unchanged and the program returns to the previous menu. When <F1> is pressed anywhere in updating the items, the program immediately returns to the previous menu without saving the changes. If <F2> is pressed at the start of entering a field, the value of the field is entered to be zero or null (""). All the previous data are displayed in square brackets.

If the 'g' item is selected, another record which has the same name field is searched. If there exists such a record, the items of that record is selected, otherwise the program gives a message :"no other person with this name" and

returns to the topmost menu.

Upon selecting the 'q' item, the program directly returns to the first menu.

In the first menu, the next option is

2) Enter a new record.

If this item is selected, similar things as in the modification operation happen. The name, ID number, address, etc. are asked and the operation stops with <ESC> or <F1> like in the update case. The only difference is that the "name" field is asked and the previously stored data is not displayed.

Another option is

3) Go to the image selection menu.

If this option is selected, the VIEW.EXE is executed without passing any list and the

Press < F# > to add/delete to/from the list

options are disabled.

The option

4) Change the data file name [TheFile]

asks for a file name to consider as the data file. The default name is "DAT.FIL" and can be changed with this option, however the written named file must be prepared and ready. The way to prepare an empty data file is to use the option

5) Create a data file

Upon selecting this option, a warning indicator is displayed :

You are about to create a data file of 2000 records.

Please type the name of this data file.

If you press <ESC> or <ENTER>, the data file will be

created with that file name (if not empty).

If you press <F1>, this operation will be cancelled.

File name:

When the user enters a file name, the program generates an empty record file with the given name.

The last option is the

Q) Exit program

option which exits the MIW.EXE program.

Chapter 5

CONCLUSIONS

In this thesis, the implementation of a PC based medical image workstation is described. The two chapters *Chapter 2* and *Chapter 3* concerned about the image processing - image viewing and data base handling stuff. We presented the details of the developed software by using flow-charts and graphical illustrations.

It is more suitable to consider other utilities such as scanning the image, Write Once Read Many (WORM) optical disk interfacing and JPEG compression - decompression in a separate chapter dedicated for experimental results. The speed considerations about displaying and data-base management are also discussed as an experimental result in the fourth chapter. The two sections "compiling the source code" and "using the program" of the fourth chapter can be used as a *"Users Manual"* for this software. A new user can read the section about using the program for learning the first steps for running the program.

The menus of the program are quite explanatory and guiding all through the steps of the different utilities, so even a person without any knowledge about the commands can use this program by following the instructions. In fact, the general viewing and image processing utilities are suitable for lots of other applications which require image scanning, storing and viewing. The effort for increasing the resolution and pixel depth of the display was motivated by the need for a high definition display in medical imaging, however, a good definition display is suitable for many other applications too.

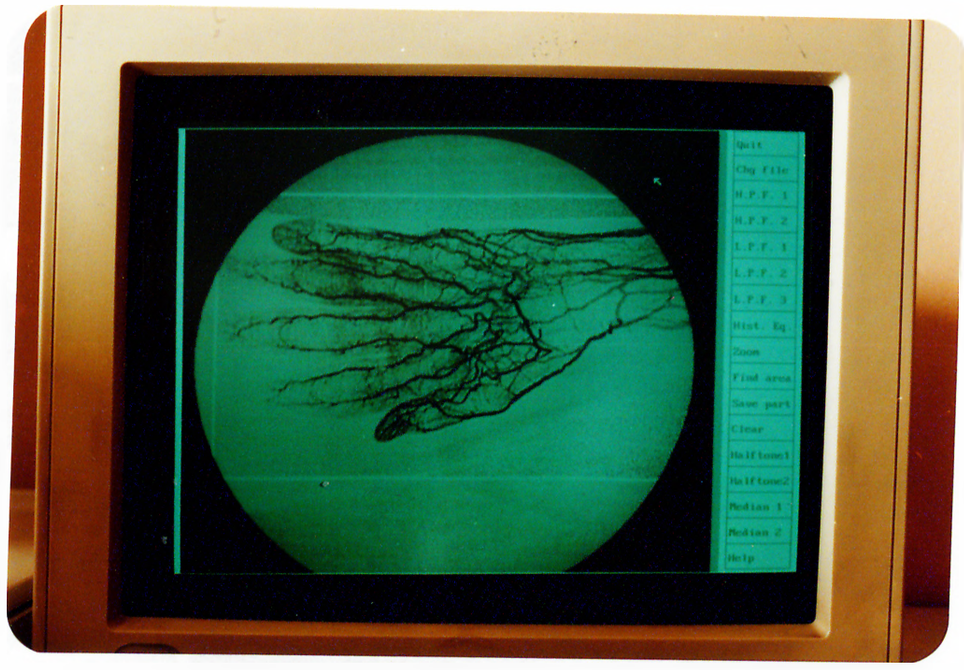


Figure 5.1: A sample image on our "medical image workstation" screen.

The following figure shows a general view of our Medical Image Workstation and the next three figures are sample photos showing the display of the Medical Image Workstation in different forms.

We implemented basic image processing operations and drawing techniques in this program. These operations together with the given peripheral devices are sufficient for personal radiological imaging purposes, however the platform of the system can be enlarged for general usage. As a matter of fact, a better imaging environment could use the "Multi-Media" concept. In that case, a centralized storage unit is connected to host computers of different kinds, having different viewing tools. The only thing to take into account is the data flow protocol. In multi-media systems, the data security and data corruption are the two important concepts to consider.

Unfortunately, both the development and the maintenance of this kind of a system is difficult and beyond the scope of a practicing radiologist. We have described the major difficulties of implementing a large system in the first chapter. A possible work can be done by a group of people from different platforms

Figure 5.2: The total view of the medical image workstation

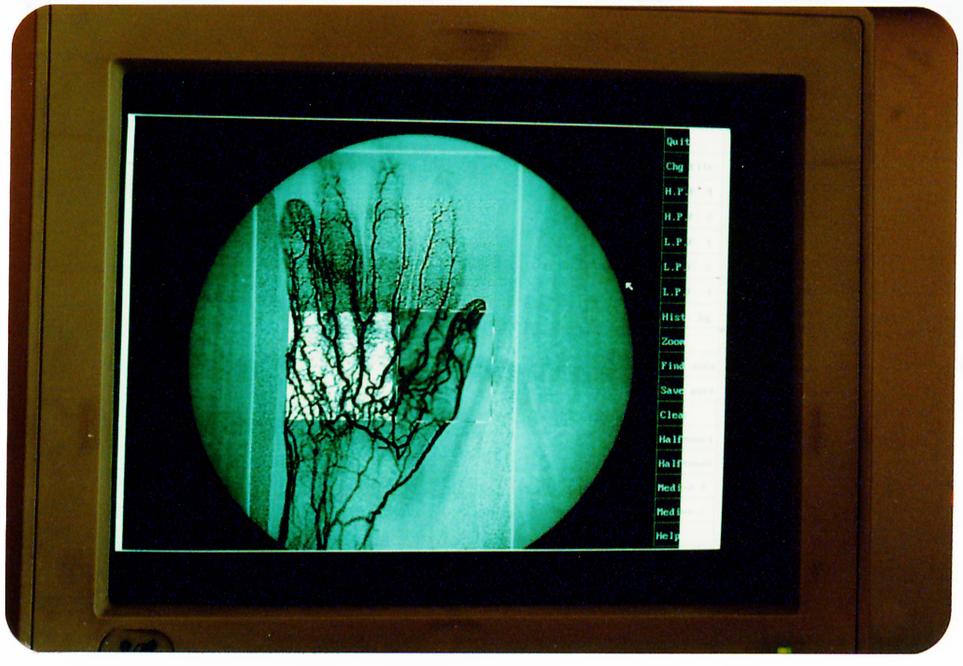


Figure 5.3: A sample image display with four operations performed on four windows. From top-left with clockwise order, a)High-pass filter (weak), b)High-pass filter (strong), c)Local histogram equalization, d)Gaussian-shape low-pass filter

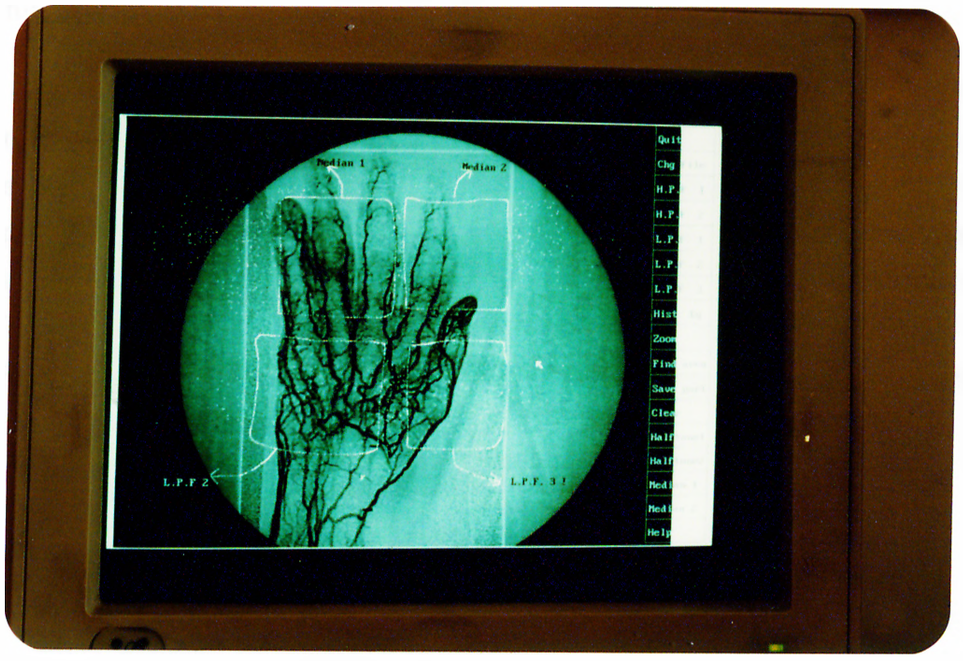


Figure 5.4: A sample salt and papper noise added image display with four operations performed on four windows. From top left with clockwise order, a)3x3 median filtering, b)Horizontal + vertical median filtering, c)Low-pass filter with neighbour averaging, d)Low-pass filter with 3x3 averaging.

such as "radiology", "software engineering", "electrical engineering with proffeciency in image processing", "electrical engineering with proffeciency in image coding and compression", "computer science with proffeciency in multi-media" and "engineering with professiency in computer networks".

With the present configuration, the utilities about image processing can be extended and a gray tone printer can be inserted. As an example for extendibility, the amount of the 3x3 linear filters can be increased and some other filtering operations can be inserted in the program.

A good feature is that our software does not cause problems with color monitors and displays the image in a uniform gray tone. Furthermore, the extension for displaying 256 color images is easy because the TIFF format supports the existence of color look-up table in the image file. Since our scanner is a gray tone scanner, we did not insert the color display module in the program. The extensions that we describe above do not cause problems in terms of the flexibility of the software. A practicing person can read this thesis and do the changes in the software in an easy manner.

Appendix A

Image Compression

In this appendix, the still picture coding standard JPEG will be described and the modifications we made will be presented

A.1 JPEG

A.1.1 File format

The JPEG compressed image file consists of the following kind of hierarchical syntax :

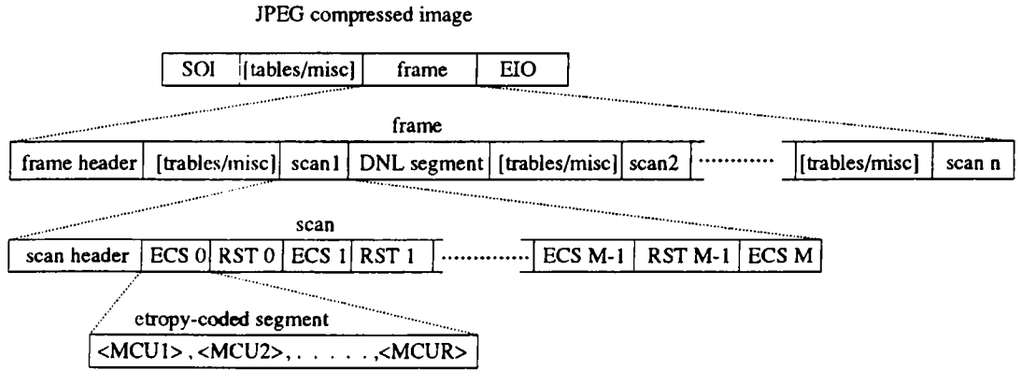


Figure A.1: File format of JPEG

The markers shown in the figure are defined as follows:

SOI : start of image marker. It shows the start of a compressed image represented in the JPEG format.
EOI : end of image marker for the same image.
DNL : define number of lines marker. It shows the line count of the next scan.
ECS : entropy coded segment which is the real data.
RST : restart marker. It is an optional switch used when restart is enabled.
MCU : minimum coded unit. It is the smallest group of data.

This hierarchical representation has lots of safety headers and unused places for future usage. It holds both DCT sequential and progressive coding scheme.

The frame header consists of six basic indicators and four component specification parameters for each component (1 component for gray-scale images, 3 components for color images). The basic indicator starts with the SOF (start of frame) marker which indicates that the image has one of the following form:
-Baseline sequential,
-Extended sequential, Huffman coding,
-Progressive, Huffman coding,
-Lossless (sequential), Huffman coding,
-Extended sequential, arithmetic coding,
-Progressive, arithmetic coding.
The second indicator is the frame length and it specifies the length of the header. The third is the sample precision. For DCT processes, this number should be either 8 or 12. Next comes two indicators for the width and height of the image. The last basic indicator specifies the number of components in frame. After these indicators, we have the component specification parameters which include one component identifier, one horizontal, one vertical sampling factor and one quantization table selector for each component.

The scan header also has a similar syntax. It has a SOS (start of scan) indicator followed by the length indicator for the scan header. Then comes the scan component selector. After these, we have the DC and AC entropy table selectors for the DCT transformed coefficients. We also have four parameters for specifying the first and last DCT coefficients that should be contained in any block of the current scan, and specifying the point transform used in the first two coefficients. These last parameters are not used for sequential DCT process (our case).

The tables and miscellaneous parameters usually consist of quantization, Huffman or arithmetic conditioning tables, restart definitions, comments and application data.

A.1.2 General description of JPEG encoding and decoding process

The International Standard specifies two classes of encoding and decoding processes, lossy and lossless. The DCT based processes are lossy and they allow high visual fidelity for the reconstructed image. At the same time they produce considerably high compression rates.

The simplest DCT based coding is the baseline sequential process. In our applications, we used this kind of coding which is sufficient for many applications. There are other DCT based coding techniques as described during explaining the SOF (start of frame) header.

The second class is not based on the DCT and it is not implemented in our application. The general encoding and decoding processes are given in figures A.2 and A.3 respectively.

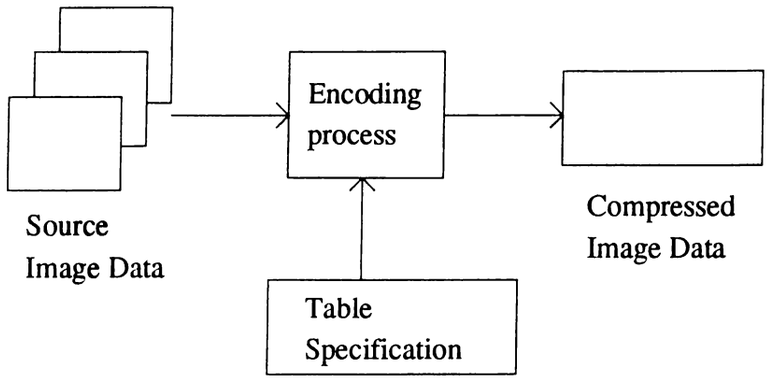


Figure A.2: General encoder

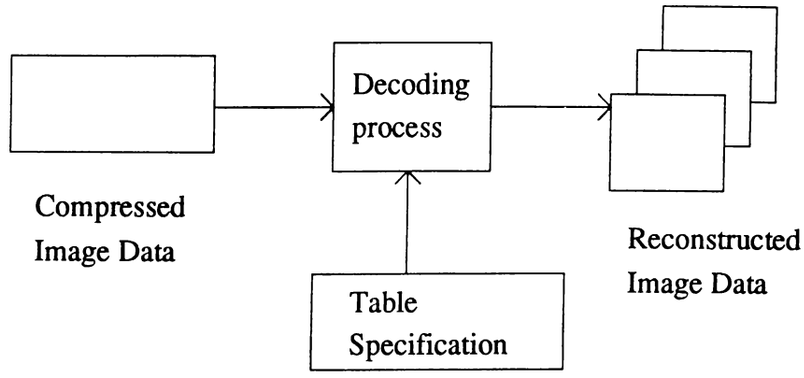


Figure A.3: General decoder

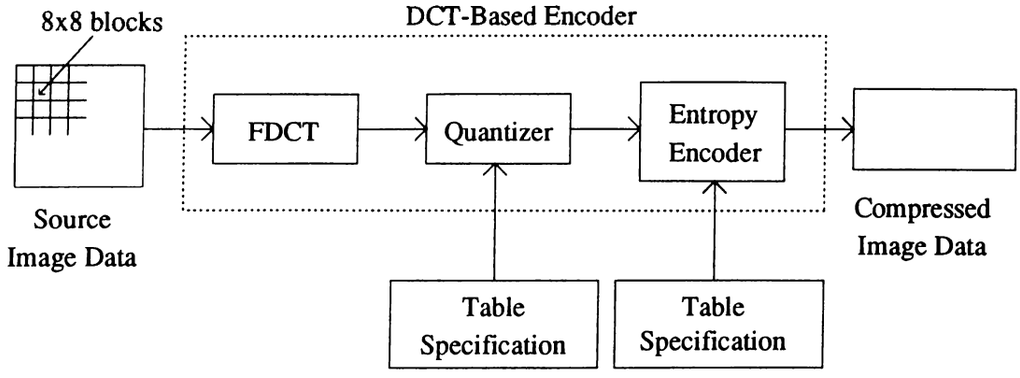


Figure A.4: DCT-based encoder diagram

A.1.3 DCT-based coding

Figure A.4 shows the main procedures for DCT based encoding. In the figure, the encoding of a single-component image is illustrated for schematic simplicity. If there is a color image composed of three components, then each color component is processed independently.

In the encoding process the input image samples are grouped into 8x8 blocks and each block is transformed by the forward DCT (FDCT) [19] into a set of 64 values, namely DCT coefficients. The first of these coefficients is called the DC value and other 63 coefficients are called the AC coefficients.

These 64 coefficients are quantized according to a quantization table which has no restricted default. After quantization, the DC coefficients of 8x8 blocks are difference coded and the other 63 AC coefficients are prepared for entropy coding by being converted into a one dimensional zig-zag sequence, as shown in figure A.5.

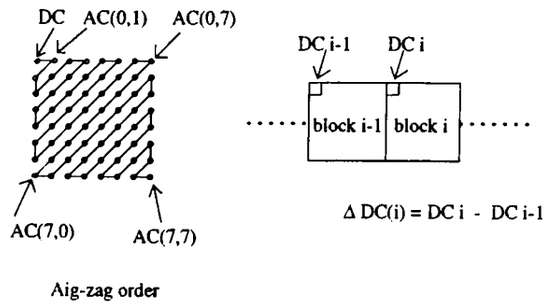


Figure A.5: DCT-based encoder diagram

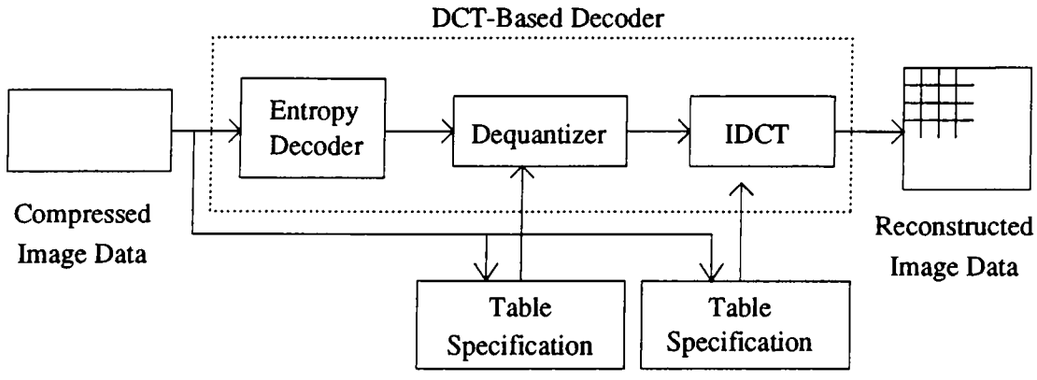


Figure A.6: DCT-based decoder diagram

All of these quantized coefficients are then entropy encoded. For our specific case, the entropy coding consists of Huffman coding. The details of Huffman code-book generation is described in detail in [18] pp C-1. But typically it is an ordinary Huffman coding scheme with some restrictions on the size of the code lengths.

After inserting the data between the appropriate headers of the JPEG file, the compressed image file becomes ready.

The DCT based decoding of the compressed JPEG file (figure A.6) is just the reverse of the encoding scheme. The entropy decoder decodes the zig-zag sequence of quantized DCT coefficients. After de-quantization, the DCT coefficients are transformed to 8x8 block of samples by the inverse DCT (IDCT) [19]. The de-quantization step obviously cannot recover the floating point precision of the originally calculated data.

A.1.4 Modes of operation

There are four distinct modes of operation defined for various coding processes: *sequential DCT-based, progressive DCT-based, lossless, and hierarchical.*

For our implementation which is the sequential DCT-based mode, the previously defined operations are performed. This is the same for the progressive DCT-based mode, but the 8x8 blocks are encoded in multiple scans through the image. There are two methods of partially encoding the blocks within a scan. In the first method, only a specified band of coefficients from the zig-zag sequence need be coded. This method is called *spectral selection* because each band typically contains coefficients which occupy a higher or lower part of the frequency spectrum. This is a result of the spectral property of the DCT [19]. The second method is called the *successive approximation* because in that case, the N most significant bits can be encoded first, then less significant bits may be encoded successively. The number N is specifiable here. Both of these methods can be used separately or mixed.

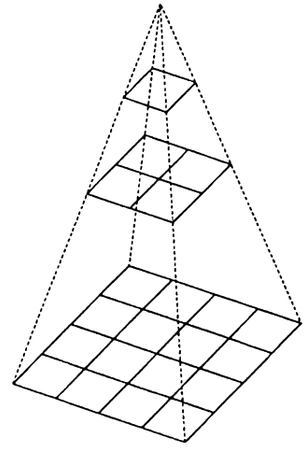


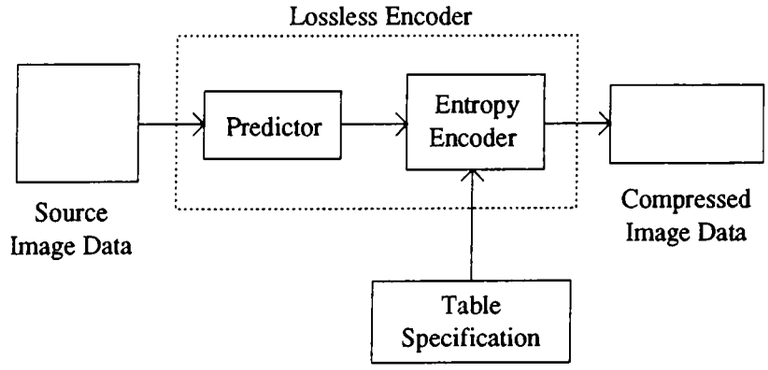
Figure A.7: Hierarchical multi-resolution encoding

In hierarchical mode, the image is coded as a sequence of frames of scaled sizes. Each smaller frame provides a reference for the larger, so at a mid level, the frame provides *reference reconstructed component* which is needed for prediction of subsequent frames. This approach is also known as *pyramidal coding.* The coding of differences between actual and predicted values may be done using only DCT based processes, only lossless processes, or DCT based processes with final lossless process for each component. Down-sampling and

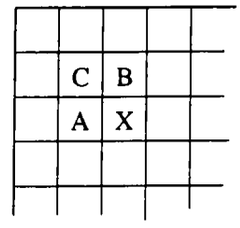
up-sampling filters may be used to provide a pyramid of spatial resolutions (figure A.7). These two methods are not implemented often for image compression purposes because of their high complexity and non-standardization.

In the lossless encoding process (A.8-(A)), a predictor combines the values of up to three neighborhood samples (A, B and C in figure A.8-(B)) to form a prediction of the sample (X in figure A.8-(B)). This prediction is then subtracted from from the actual value of sample X, and the difference is entropy coded.

The baseline sequential process uses Huffman coding, but the extended DCT based and lossless processes may use either Huffman or arithmetic coding.



(A)



(B)

Figure A.8: Lossless encoder process

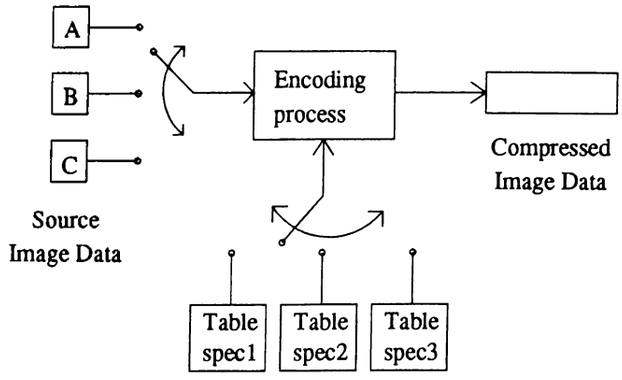


Figure A.9: Component-interleave and table-switching control

A.1.5 Multiple component control

In our software, we supported gray-scale images which has a single component, but the general JPEG standard requires handling the color components. The basic point is that, in any method, the correct table specifications must apply to the correct image component (figure A.9).

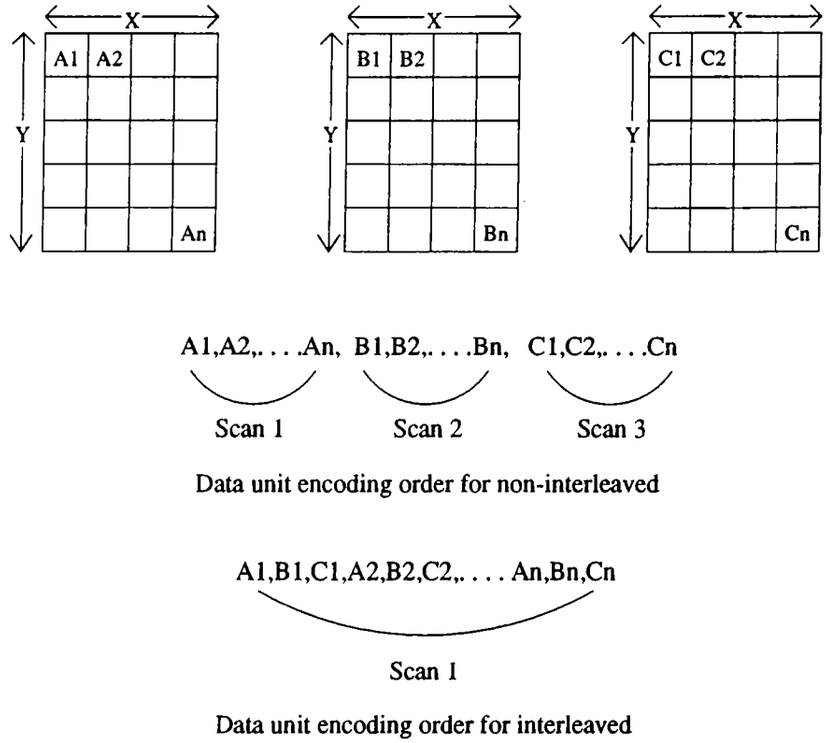


Figure A.10: Interleaved versus non-interleaved encoding order

In sequential mode, the encoding is non-interleaved if the encoder compresses all the image data units in component A before beginning component

B, and all in B before C. Conversely, the encoding is interleaved if the encoder compresses a data unit from A, a data unit from B, a data unit from C, and then back to A, and so on. These two cases are illustrated in figure A.10. If the image components have different dimensions, the sequential mode has no problem, and the interleaved mode handles the situation by putting proportional amount of data units to the coded stream.

A.1.6 Discrete Cosine Transform

Before the encoder computes the forward DCT for a block, the samples must be level shifted to a signed representation by subtracting 2^{p-1}, where "p" is the precision (8 or 12). This shift is necessary to compute proper DCT values.

The forward and inverse discrete cosine transforms for 8x8 blocks are defined in the following way :

$$FDCT : S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)u\pi}{16} \qquad \text{(A.1)}$$

$$IDCT : s_{yx} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{vu} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)u\pi}{16} \qquad \text{(A.2)}$$

where $C_u, C_v = 1/\sqrt{2}$ for $u, v = 0$; $C_u, C_v = 1$ otherwise.

The precision requirements for implementing the numbers like π in the encoder and decoder are also determined by the JPEG standard.

There is no required or recommended method for implementing these 8x8 DCT's.

A.1.7 Quantization of the DCT coefficients

After the FDCT is computed for a block, each of the 64 DCT coefficients is quantized by a uniform quantizer. The value of the coefficient S_{vu} is divided by the corresponding quantization element Q_{vu} from the defined quantization table. The uniform quantizer is defined as:

$$Sq_{vu} = \text{round}\left(\frac{S_{vu}}{Q_{vu}}\right) \qquad \text{(A.3)}$$

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	52	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	104	99

Table A.1: Luminance quantization table

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Table A.2: Chrominance quantization table

The reverse quantization is the inverse everse of this operation on the average:

$$R_{vu} = Sq_{vu} \times Q_{vu} \tag{A.4}$$

The two examples above are typically successful for 8 bits per sample Y, U, V images transformed with the NTSC standard.

These are only example values and there are no restrictions on the quantizer elements because these tables should be included in the coded bit stream.

A.2 Transform matrices other than DCT

In our research, we especially dealt with gray tone radiological images. It is known that JPEG is a good standard for still image compression on the average of pictures, but one could also experiment the effects of the transformation in JPEG coding.

For this purpose, we tested a new class of block transforms. These transforms are constructed from subband decomposition filter banks corresponding to regular wavelets and the transformation is called the "BLOCK WAVELET TRANSFORM" [20], [21]. In this section, we will present the BWT techniques.

The idea of block wavelet transform comes from the fact that frequency domain waveform coding methods have been widely used in practice [3]. In these methods one takes the advantage of nonuniform distribution of bits to frequency components. Block transform coding and subband coding methods are the most popular frequency domain waveform coding techniques [3], [22].

In subband image coding the frequency domain is divided into typically four or more subbands by a filter bank [23]. Each subband is down-sampled at its Nyquist rate which is twice the width of the band and bits are judiciously allocated to the subband signals. In image coding the multi-rate (multi-resolution) signal representation allows the use of embedded or hierarchical coders. In this way the nonuniform distribution of energy in the frequency domain is exploited. This is also the case for DCT.

In almost all transform coding methods the signal is first divided into blocks or vectors [23] [24] [25]. These blocks are linearly transformed into another domain. The resultant set of transform domain coefficients (or frequency components) are then quantized for transmission or storage. The efficiency of the transform coding system depends on the type of linear transform and the nature of bit allocation to the transform domain coefficients. The most popular linear transform is the Discrete Cosine Transform (DCT). Image and video coding standards employing DCT are developed and widely used [24] [25].

Recently, the strong relation between wavelet theory [25] [26] and subband decomposition is established [27] and it is shown that the wavelet orthonormal bases serve to provide a useful multi-resolution signal representation. Corresponding to each wavelet orthonormal basis there exists a subband decomposition filter bank realizing the multi-resolution signal representation. It is shown in [20] and [21] that a new class of linear block transforms can be obtained from subband based multi-resolution signal decomposition. The new class of block transforms, Block Wavelet Transforms (BWT), are constructed from subband filter banks corresponding to regular wavelets [26].

In our experiments, we observed that the "scalability" [23] property of some specific BWTs are very good. In this scheme, 8x8 image sub-blocks are transformed by a BWT. The low-resolution images recovered from the first 4x4 BWT coefficients are compared to the low-resolution signal which can be extracted from a DCT based scheme. These comparisons showed that some BWT matrices perform better than DCT in terms of "scalability".

The compression capability is the main concern, so BWT is compared to DCT in terms of compression capability too.

A.2.1 BWT matrix construction

Let us assume that $H_0(\omega)$ and $H_1(\omega)$ are the low-pass vand high-pass filters of a perfect reconstruction filter bank, respectively as shown in Fig. A.11. In a two band partition the input signal $x[n]$ is filtered by $h_0[n]$ and $h_1[n]$ and the resultant signals are down-sampled by a factor of two. In this way two sub-signals $x_0[n]$ and $x_1[n]$ are obtained, i.e.,

$$x_i[n] = \sum_k h_i[k]x[2n-k], \quad i = 0, 1 \tag{A.5}$$

The sub-signal $x_0[n]$ ($x_1[n]$) contains the low-pass (high-pass) information of the original signal $x[n]$. In many signal and image coding methods this signal decomposition operation is repeated in a tree-like structure and the resultant sub-signals are compressed by various coding schemes [22] [23].

Let us first construct the BWT for a vector of size $N = 2$. We assume that the filters $h_0[n]$ and $h_1[n]$ are FIR perfect reconstruction filter pairs [26], [27] corresponding to regular wavelets. Let $x[n]$ be a finite extent signal of duration $N = 2$. The signals, $x[n] * h_0[n]$ and $x[n] * h_1[n]$ have durations $K + N - 1 = K + 1$, if the durations of the filters, $h_0[n]$ and $h_1[n]$, are K. In spite of down-sampling by two $x_0[n]$ and $x_1[n]$ contain more than two samples in order to achieve perfect reconstruction.

If $H_0(z)$ and $H_1(z)$ are IIR low-pass and high-pass filters of a two-channel perfect reconstruction filter bank, the number of samples become infinity.

Let us define a periodic sequence $\tilde{x}[n]$ as a periodic extension of $x[n]$, i.e, $\tilde{x}[n] = \sum_m x[n + Nm]$. For $N = 2$, the signals $\tilde{x}[n] * h_0[n]$ and $\tilde{x}[n] * h_1[n]$ are

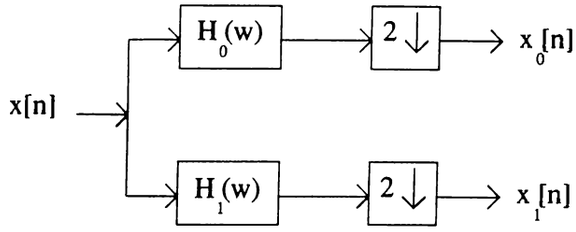


Figure A.11: 1 stage subband filtering

also periodic signals with period $N = 2$. Therefore, the sub-signals, $x_0[n]$ and $x_1[n]$ are periodic signals with period $N/2 = 1$. We define $x_0[0]$ and $x_1[0]$ as BW transform domain coefficients of the vector $\mathbf{x} = [x[0] \;\; x[1]]^T$. Clearly, $x_0[0]$ ($x_1[0]$) contains the low-pass (high-pass) frequency information of the vector $\mathbf{x}$. Since convolution and down-sampling operations are linear, this transform is also a linear transform. Let us call the transform matrix $\mathbf{A}_2 = [a_{ij}]_{i,j=1,2}$. The entries of the 2×2 BWT matrix can be determined by using the basis vectors $\mathbf{e}_1 = [1 \;\; 0]^T$ and $\mathbf{e}_2 = [0 \;\; 1]^T$. This is equivalent to applying the periodic signals, $\tilde{e}_1[n] = \sum_m \delta[n + 2m]$ and $\tilde{e}_2[n] = \sum_m \delta[n - 1 + 2m]$ to the subband decomposition filter bank. When the input is $\tilde{e}_1$ then the output of the upper (lower) branch is a_{11} (a_{21}). Similarly, a_{11} and a_{22} are obtained by using $\tilde{e}_2$.

Let us now construct the BWT matrix for $N = 2^\ell$ where ℓ is a positive integer. Let $x[n]$ be a finite extent signal of length N and $\mathbf{x} = [x[0]\, x[1] \ldots x[N-1]]^T$ be a vector of size N. In this case we consider an N band partition of the frequency domain. In order to achieve such a partition one can use the filter bank of Figure A.11 in ℓ stages as shown in Figure A.12.

This ℓ stage operation is equivalent to a single stage operation consisting of filtering $x[n]$ by the convolution of filters on the signal path and down-sampling the filtered signal by $2^\ell = N$. The frequency response of the total convolved filter $h^0[n]$ is given by

$$H^0(\omega) = H_0(\omega)H_0(2\omega)...H_0(2^{\ell-1}\omega) \tag{A.6}$$

In general the frequency response of $h^l[n]$ in an N stage partition is given by

$$H^l(\omega) = H_{i_0}(\omega)H_{i_1}(2\omega)....H_{i_{N-1}}(2^{\ell-1}\omega), \quad l = 0, 1, ..., N-1 \tag{A.7}$$

where $l = i_0 2^{\ell-1} + 2^{\ell-2} i_1 + ... + i_{N-1}$. When the finite signal, $x[n]$, is applied to the filter bank of Fig. A.12, in spite of down-sampling by N, the sub-signals $x_0[n]$, $x_1[n], ..., x_{N-1}[n]$ contain more than N samples due to filtering. Also,

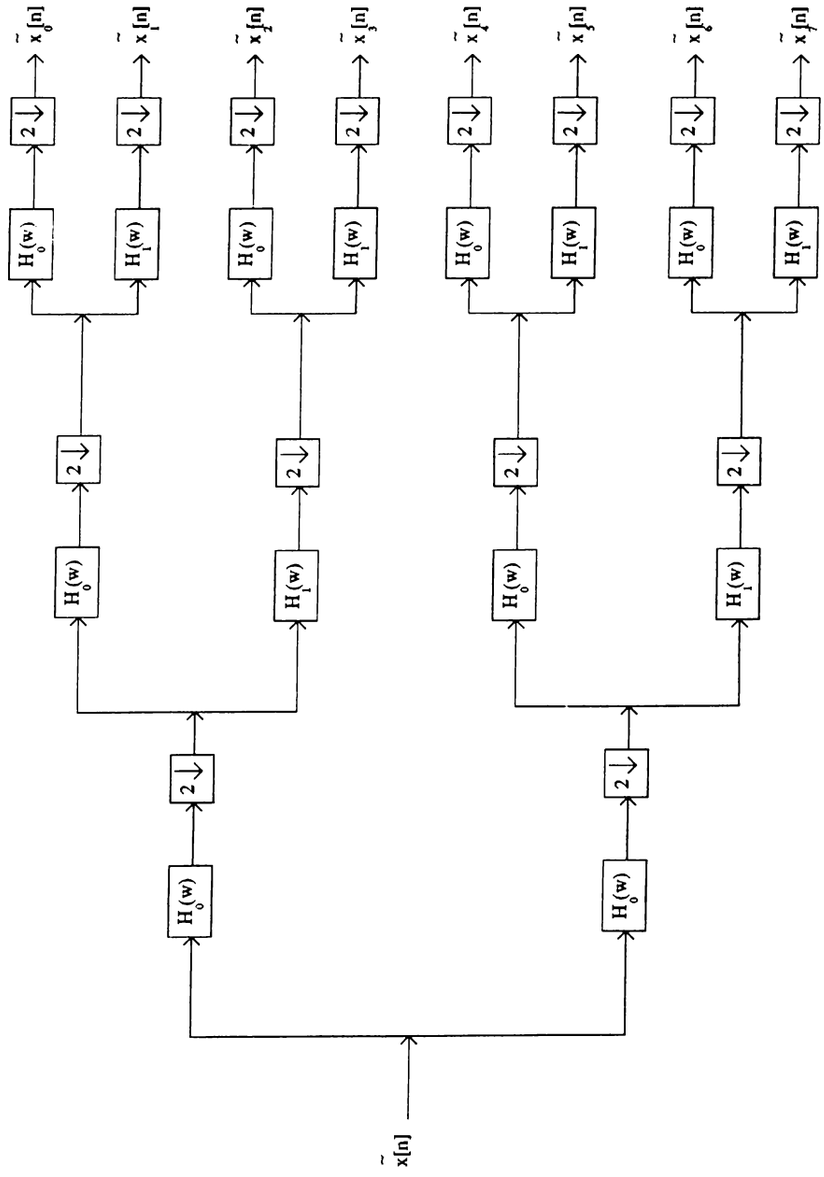


Figure A.12: 3 stage subband filtering

to achieve perfect reconstruction one needs more than N samples for practical filter banks. Let $\tilde{x}[n] = \sum_{m=-\infty}^{\infty} x[n+Nm]$. Since $\tilde{x}[n]$ is a periodic signal with period N the sub-signals, $\tilde{x}_l$, are also periodic with period 1. As in the case of $N = 2$ we define the BW transform vector, $\theta = [\theta_0\ \theta_1 ... \theta_{N-1}]^T$, as follows

$$\theta_l = \tilde{x}_l[0], \quad l = 0, 1, ..., N-1 \tag{A.8}$$

The j-th column of the $N \times N$ transform matrix $\mathbf{A}_N = [a_{ij}]_{i,j=1,2...,N}$ can be obtained by applying $\tilde{e}_j[n] = \sum_m \delta[n-(j-1)+Nm]$ as the input to the N-band subband decomposition structure. In this case the branch outputs are periodic signals with period 1 and the value at the i-th branch is the (i,j)-th entry, a_{ij}, of the matrix, $\mathbf{A_N}$.

Block Wavelet Transforms obtained as above are orthogonal transforms, i.e., $\mathbf{A_N A_N^T} = \mathbf{I_N}$. Fast implementation of the transform is carried out in ℓ stages by the subband-decomposition structure shown in Figure A.12. If the order of the FIR filter, h_0, is K, then it requires $2KN \log N$ (Order($N \log N$)) multiplications to get N transform domain coefficients.

A.2.2 BWT Examples

In this section we present several BWTs constructed from the Haar orthogonal basis [26], perfect reconstruction filter banks of Barnwell and Smith [28], Daubechies wavelets [26] [29] and Butterworth IIR filter banks.

Block Haar Wavelet Transform: This transform is constructed from the simplest perfect reconstruction filter bank: $h_0[0] = h_0[1] = 1/2, h_0[n] = 0$ for $n \neq 1, 2$, and $h_1[n] = (-1)^n h_0[n]$ [23]. This filter bank structure produces the Haar (wavelet) basis of $L^2(R)$ [27]. Since the filters h_0 and h_1 are second order FIR filters, the filters $h^l[n]$, $l = 0, 1, ..., N-1$ are N-th order filters. Because of this fact the rows of the $N \times N$ transform matrix $\mathbf{A}_N$ are the impulse responses of the filters $h^l[n]$ and the transform matrix is nothing but the well-known Hadamard Transform Matrix [22].

Block Daubechies Transform: Daubechies designed perfect reconstruction filter banks in [26]. These filter banks correspond to regular wavelets. We constructed the BWT matrix from Daubechies filter banks. For the filter pair, $h_0[n] = \{..., 0, .230378, .714847, .630881, -.279838, -.187035,$

$.0308414, .0328830, -.0105974, 0, ...\}$ and $h_1[n] = (-1)^n h_0[-n+1]$, the 8x8 transform matrix $\mathbf{A_8}$ is given as follows:

$$\mathbf{A_8} = \begin{bmatrix} 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 \\ 0.0309 & 0.2904 & 0.5147 & 0.3870 & -0.0309 & -0.2904 & -0.5147 & -0.3870 \\ 0.3102 & 0.3921 & -0.3102 & -0.3921 & 0.3102 & 0.3921 & -0.3102 & -0.3921 \\ -0.5147 & -0.3870 & 0.0309 & 0.2904 & 0.5147 & 0.3870 & -0.0309 & -0.2904 \\ 0.3536 & -0.3536 & 0.3536 & -0.3536 & 0.3536 & -0.3536 & 0.3536 & -0.3536 \\ -0.3717 & 0.1485 & 0.3097 & -0.4938 & 0.3717 & -0.1485 & -0.3097 & 0.4938 \\ -0.3921 & 0.3102 & 0.3921 & -0.3102 & -0.3921 & 0.3102 & 0.3921 & -0.3102 \\ -0.3097 & 0.4938 & -0.3717 & 0.1485 & 0.3097 & -0.4938 & 0.3717 & -0.1485 \end{bmatrix}$$

Block Transform obtained from Smith and Barnwell (SB) filters: We also obtained block wavelet transforms by using the perfect reconstruction filter banks described in [28].

Butterworth BWT: This transform is constructed from Butterworth filter bank:

$$H_i(z) = 1 + (-1)^i z \prod_{k=1}^{L} \frac{z^2 cot^2((\pi k/(2L+1)) + 1}{z^2 + cot^2(\pi k/(2L+1))}, \quad i = 0, 1 \qquad \text{(A.9)}$$

The BWT matrix $\mathbf{B_8}$ is given as follows:

$$\mathbf{B_8} = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1.4 & 1 & 0.2 & -1 & -1.4 & -1 & -0.2 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & 0.2 & -1 & -1.4 & -1 & -0.2 & 1 & 1.4 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -1.4 & 1 & -0.2 & -1 & 1.4 & -1 & 0.2 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & -0.2 & -1 & 1.4 & -1 & 0.2 & 1 & 1.4 \end{bmatrix}$$

Note that the BWT matrices are orthogonal.

Simulation examples are presented in the next section.

A.2.3 BWT simulation examples

Consider an $AR(1)$ random process with $\rho = 0.95$. Fig A.12 shows the basis restriction error [22] of the transform coefficients for Karhunen-Loeve Transform

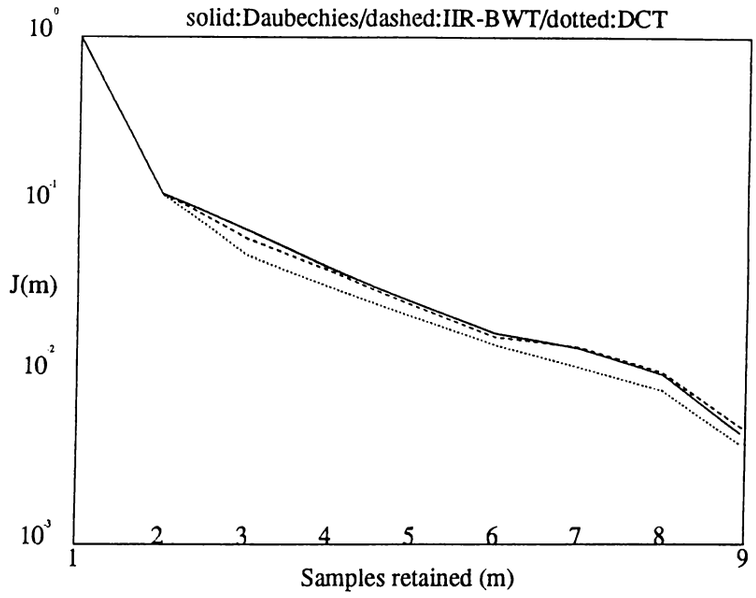


Figure A.13: Basis restriction error figure (J(m) = $\sum_{k=m}^{N-1} \sigma_k^2 / \sum_{k=0}^{N-1} \sigma_k^2$)

(KLT), DFT and BWTs obtained from Butterworth and Daubechies [28] filter banks. It can be observed from Fig A.13 that the performance of Daubechies BWT and Butterworth BWT's are very close to each other.

We inserted the BWT matrices into the JPEG program instead of the DCT module. The following are the obtained results.

When the signal to noise ratio of the reconstructed Barbara (672x560 with 8 bit/pel) test image, which has a high spatial entropy is considered, the DCT compressed to 1.349 bit/pel, the Daubechies BWT compressed to 1.487 and the Butterworth BWT compressed to 1.440 bit/pel. The reconstructed images were all visually indistinguishable from the original.

A 512 x 512 abdominal CT image with 10 bit/pel can be compressed to 2.2 bit/pel with DCT, 2.22 bit/pel with the Butterworth BWT and 2.23 bit/pel with the Daubechies BWT. The reconstructed images had SNRs very close to each other, 42.13dB, 42.10dB and 42.17dB respectively.

For the high detailed radiology images, the BWTs compressed very similar to DCT based JPEG and for some specific images, BWTs defeated DCT, however for the low resolution X-ray images, DCT performed better.

After these experimental results and taking into account that the DCT is a well established standard, we decided to use DCT in the JPEG compression software.

Appendix B

Summary of the Tag Image File Format (TIFF) Revision 5.0

We will present a brief specification of the TIFF image format. This information is based on [1, p.459] where it is written that the specifications were obtained from © Aldus Corporation.

B.1 Structure

The individual fields in TIFF are indicated by unique tags. In this way, one can omit or include the information about images in an application. A TIFF file starts with an "image file header" which is 8 bytes long. This header points to one or more image data in which the information about the images and the images themselves. This pointed place is called "image file directory, IFD".

The header begins with :
Bytes 0,1 : "II" (hex 4949) or "MM" (hex 4D4D) where for "II" the byte order is from least significant to most significant and for "MM" the case is vice versa. The Intel processors use the "II" type.
Bytes 2,3 : TIFF version number : 42 (hex 2A) always.
Bytes 4-7 : The offset of the first Image File Directory (IFD). The directory may be at any location in the file after the header but must begin on a word (two bytes) boundary.

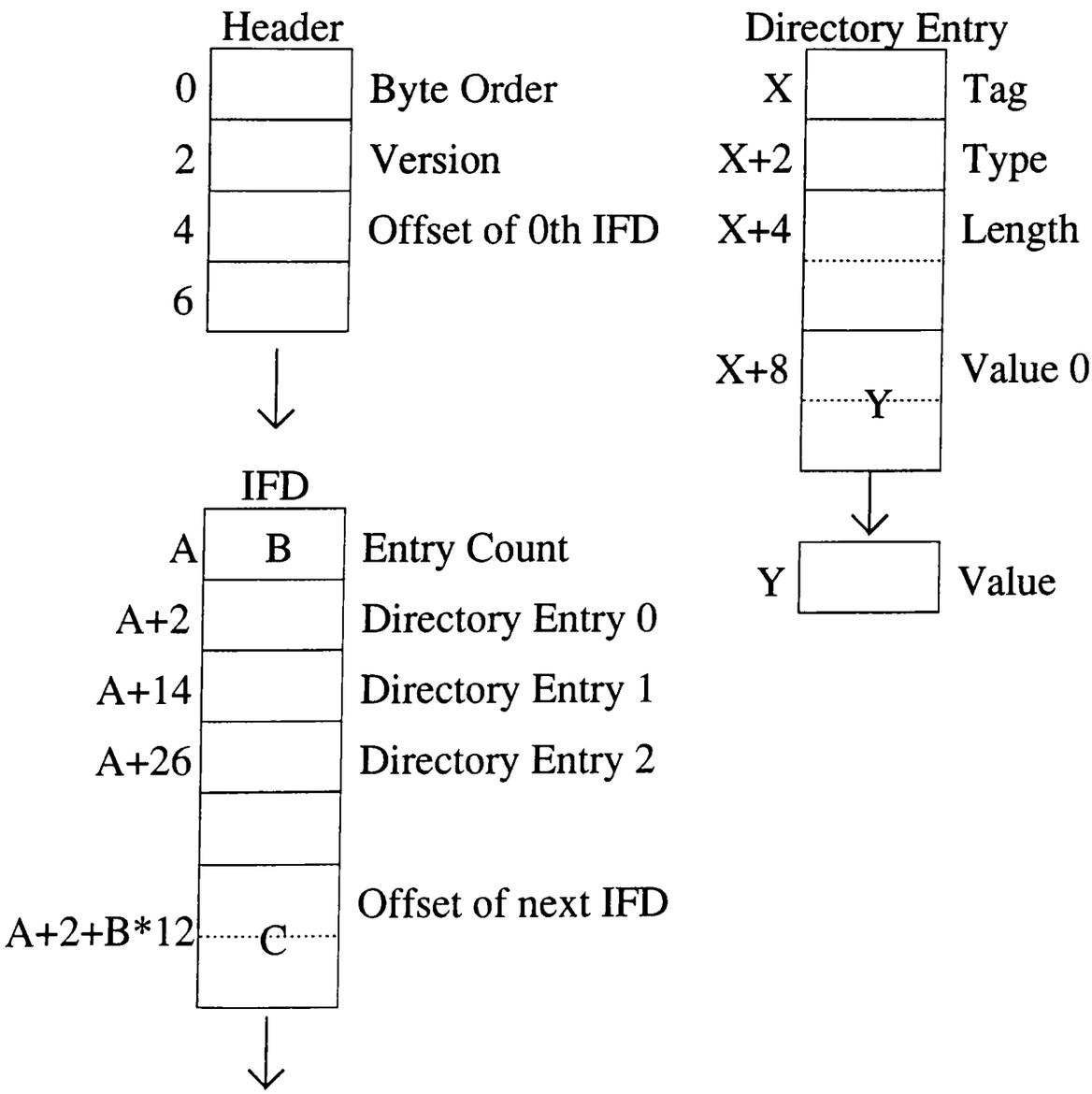


Figure B.1: The global structure of TIFF

This pointed IFD consists of a 2-byte count of the entries (this is the number of fields in the IFD), followed by a sequence of 12-byte field entries and a 4-byte offset of the next IFD. If there is no next IFD, 4 bytes of 0 is put. The 12-byte IFD entry is as follows :
Bytes 0,1 : the Tag for the field.
Bytes 2,3 : the field type.
Bytes 4-7 : the length of the field.
Bytes 8-11 : the file offset of the "value" for the field.

Any application to read a TIFF file must read the elements of the IFD entry in sequential order to go to a specific field. If a field is to be skipped, its field length must be read first and then that much of bytes can be skipped. This kind of approaches are very flexible for general usage.

B.2 Basic fields

We will present the basic fields that we used in reading the TIFF files with our configuration.

BitsPerSample
-Tag : 258
-Type : SHORT (16 bits)
-N (number of values) : SamplesPerPixel
-explanation : 8 for gray-scale, 8,8,8 for full RGB, 1 for bitmap, etc.

Compression
-Tag : 259
-Type : SHORT
-N : 1
-explanation : 1 for no compression, but pack data, 2 for run length encoding, 5 for LZW compression.

GrayResponseCurve
-Tag : 291
-Type : SHORT
-N : $2^{\text{BitsPerSample}}$

-explanation : Stands for the gray tone lookup table.

ColorMap

-Tag : 320

-Type : SHORT

-N : $3 \times 2^{BitsPerSample}$

-explanation : Stands for the color lookup table.

GrayResponceUnit

-Tag : 290

-Type : SHORT

-N : 1

-explanation : 1 for tenths of a unit, 2 for hundredths of a unit, etc. Modifies the GrayResponceCurve.

ImageLength

-Tag : 257

-Type : SHORT or LONG (16 or 32 bits)

-N : 1

-explanation : The image's vertical (Y) length in pixels

ImageWidth

-Tag : 256

-Type : SHORT or LONG

-N : 1

-explanation : The image's horizontal (X) width in pixels

SamplesPerPixel

-Tag : 277

-Type : SHORT

-N : 1

-explanation : This number is 1 for bilevel, gray-scale and palette color images, it is 3 for RGB images.

There are many more (totally 45) TIFF tags for general usage, but we used this much of information to determine the sufficient information about the TIFF file. Actually, the scanner produces the gray scale image with 8 bits per pixel, but we must take care about some other fields in the case of a need

for viewing images produced by a different source.

The "PackBits" and "LZW" compression requirements are described in pages 493 and 505 of [1], however, we did not implement the decoder for these coding schemes. The performance of JPEG is preferred for image compression.

As a result of reading the SamplesPerPixel, BitsPerSample and ColorMap values, we may decide that the image belongs to one of the four TIFF classes,

. Class B for bilevel (1-bit) images
. Class G for gray-scale images
. Class P for palette color images
. Class R for RGB full color images

Our program accepts only Class P images. Because of this, the Tag fields related to other classes are skipped in our software during getting the image information.

REFERENCES

[1] Craig A. Lindley, *Practical Image Processing in C*, John Wiley and Sons, 1991

[2] M. Rabbani and P. W. Jones, *Digital image compression techniques*, SPIE Press, 1991.

[3] H. K. Huang, O. Ratib, A. R. Bakker, and G. Witte (Ed.), *Picture archiving and communication systems (PACS) in medicine*, Springer-Verlag, Berlin Heidelberg, 1991

[4] George Diehr, Terry Barron, Thomas Munro, *Basic Programming for the IBM Personal Computer*, John Wiley and Sons, 1987

[5] Paul S. Cho, Osman Ratib, *Personal digital image filing system*

[6] Richard Wilton, *Programmers Guide to PC and PS/2 Video Systems*, Microsoft Press, 1987

[7] Michael Hyman, *Advanced DOS*, Mis:Press, 1989

[8] Herbert Schildt, *C the Complete Reference, Second Edition*, McGraw - Hill, 1990

[9] Herbert Schildt, *Turbo C/C++ the Complete Reference*, McGraw - Hill, 1990

[10] Walter A. Triebel, *The 386DX Microprocessor*, Prentice - Hall, 1992

[11] Jae S. Lim, *Two Dimensional Signal and Image Processing*, Prentice - Hall, 1990

[12] Rafael C. Gonzales, Paul Wintz, *Digital Image Processing, Second Edition*, Addison - Wesley Publishing Company, 1987

[13] Anil K. Jain , *Fundamentals of Digital Image Processing*, Prentice - Hall, 1989

[14] Alan V. Oppenheim, Ronald W. Schafer, *Discrete-Time Signal Processing, Second Edition*, Prentice - Hall, 1989

[15] George B. Thomas Jr., Ross L. Finney, *Calculus and Analytic Geometry, Sixth Edition*, Addison - Wesley Publishing Company, 1984

[16] Fred R. McFadden, Jeffrey A. Hoffer, *Data Base Management*, Benjamin/Cummings Publishing Company, 1985

[17] Charles W. McNichols, Sara F. Rushinek, *Data Base Management, A Microcomputer Approach* Prentice - Hall, 1988

[18] JPEG ind. group, *Working draft for development of JPEG*, 1991

[19] N. Ahmed, T. Natarajan, K. R. Rao, Discrete Cosine Transform, *IEEE Trans. on Computers*, vol. 23, pp. 90-93, 1974.

[20] A. E. Çetin , Ş. Ulukuş , Ö. N. Gerek , *Signal Recovery from Block Wavelet Transform, Wavelets and Applications Conference*, Toulouse, June 1992.

[21] A. E. Çetin, Ö. N. Gerek, Ş. Ulukuş, *Image Coding using Block Wavelet Transform, IEEE Transactions on Circuits and Systems for Video Technology*, to appear in October 1993.

[22] N. S. Jayant, P. Noll, *Digital Coding of Waveforms*, Prentice-Hall, 1984.

[23] J. W. Woods, Ed., *Subband Image Coding*, Kluwer, 1991.

[24] Didier Le Gall, *Digital Multimedia Systems: Digital Image and Video Standards, Communications of the ACM*, Vol. 34, pp. 47-58, 1991.

[25] Y. Meyer, *Ondelettes et Operateurs*, Hermann, 1988.

[26] I. Daubechies, *Orthogonal bases of compactly supported wavelets, Commun. Pure and Applied Math*, Vol. XLI, pp. 909-996, 1988.

[27] S. G. Mallat, *A Theory for Multiresolution Signal Decomposition: The Wavelet Representation, IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol II, no 7, pp. 674-693, July 1989.

[28] M.J.T. Smith and T. Barnwell, *Exact reconstruction techniques for tree-structured subband coders*, *IEEE Transactions on Acoustics Speech and Signal Processing,* vol. 34, pp. 434-441, June 1986.

[29] R. Ansari, C. Guillemot, J. F. Kaiser, *Wavelet Construction Using Lagrange Halfband Filters*, *IEEE Transactions on Circuits and Systems*, vol. 38, pp. 1116-1118, 1991.