1995

Video coding for Bit rates 64Kbps and below

Geoffrey N. Alagoda
Edith Cowan University

Edith Cowan University

Copyright Warning

Video Coding For Bit Rates 64Kbps and Below.

By

Geoffrey N. Alagoda

A Thesis Submitted in Fulfillment of the Requirements for the Award of

Bachelor of Engineering (Computer Systems)

at the Department of Computer and Communication Engineering
Edith Cowan University

Principal Supervisor : Dr. Stefan Lachowicz

Submission Date : 3 November 1995

USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.

Table of Contents

Contents	*Page*
Declaration	5
Acknowledgments	6
Abstract	7
List of Figures	8
List of Tables	9
Key Terms	10
Chapter I : Introduction	**11**
Background	11
The Need For This Scheme	13
Related Main Problem Questions	14
Significance of Such a Scheme	14
Outline of Project with regards to Scheme	15
Chapter II : Introduction	**16**
Block Transform Coding	16
Subband Coding	17
Chapter III : The ITU-TS Recommendation H.261	**22**
Introduction	22
The Overall Compression - Decompression Process	23
The Image Format	25
Image Subdivision	27
Motion Compensation and Estimation	28
Transform (DCT & IDCT)	30
Quantisation and Inverse Quantisation	32

Zigzag Encoding and Decoding 33

Entropy (VLC - Huffman) Coding 33

Advantages of H.261 36

Disadvantages of H.261 36

H.261 Compression Performance 37

Chapter IV : Subband DCT Image Coding Approach 38

Introduction 38

The Subdivision Scheme 39

The Overall Coding - Decoding process 42

Chapter IV : The possible combination of SBDCT and Recommendation H.261 to produce a new coding scheme. 44

Introduction 44

The Overall Compression -Decompression Process 44

The Frame Size 46

The Transform 47

Motion and Estimation 48

The VLC Coding Scheme 48

The Quantising Scheme 48

Conclusion 49

Chapter VI : Project Review 50

Background 50

H.261 Implementation Specifics

(Program Documentation) 51

Improvement to H.261 54

Chapter VII : Further Improvements and Conclusions 55

Further Improvements 55

Conclusion 56

References 57

Appendix A - H.261 Encoder Source Code

Appendix B - H.261 Decoder Source Code

Declaration

I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or a diploma in any institution of higher education; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due references is made in the text.

Acknowledgments

I would first like to thank my supervisor, Stefan Lachowicz for all the help and encouragement I was given. My thanks also goes out to Andy C. Hung, who had the time to reply back to me, in regards to the H.261 scheme. To Dr. Hon Cheung for finding time to explain some difficult concepts. and to Alfred (Keng) Tan, for being a good friend and helping me through this.

Abstract

With digital motion video for either transmission or storage, there always exists a tradeoff between the data transmission rate and the picture quality. The lower the transmission bit-rate is made, the more the quality of the ima[illegible] tends to degrade. With usual transform coding schemes the degradation usually occurs when low bit-rates, that is less than about 64Kbps, are used. The resultant image tends to suffer visually from a "blocking" effect. This thesis therefore, is based on the development of a different implementation scheme, for motion video compression or encoding, so as to support both low bit-rates, around 64Kbps or below while eliminating the "blocking" effect. This scheme is designed round the compression of CIF, QCIF, and NTSC motion video frames, which are defined with three (one luminance and two chrominance) components per frame. These frame sizes are the same sizes used in the well-known ITU-TS standard called Recommendation H.261. This implementation scheme therefore closely follows that of the H.261 standard except, where such functions as the DCT transform and other modifications are needed. The Subband DCT transform used here, in replacement for the H.261 transform sub-section, is based on the work done by Yuk-Hee Chan and Wan-Chi Siu. The application of this scheme should provide similar bit-rates to that of the Recommendation H.261. However, it should also provide images free of the 'blocking' effect inherent to all encoders that spatially split the image to blocks before transform coding.

List Of Figures

Figure	Page
1 : QMF Subband Encoding - Decoding	19
2 : 4 Band image Split	19
3 : Pyramidal Structure	20
4 : Typical Functions of an H.261 device	23
5 : H.261 Overall Encoding Process	24
6 : H.261 Overall Decoding Process	24
7 : H.261 Pixel Arrangement	26
8 : Frame Subdivision	28
9 : 2D DCT Frequency Representation	31
10 : Simple Pizza Image	31
11 : Zigzag Coding - Decoding	33
12 : Quantisation Effects	37
13 : The Modified Overall Encoding Process	45
14 : The Modified Overall Decoding Process	45
15 : H.261 Implementation Test	53

List Of Tables

Table	Page
1 : Image Formats	25
2 : Transformed Pizza	31
3 : H.261 Compression	37
4 : Wasted Lines	46
5 : Image Transform Depth Levels, Widths and Heights	47

CIF, QCIF, NTSC - These are frame sizes used for the images sequences used by the Recommendation H.261.

DCT (Discrete Cosine Transform) - This is the basic transform that is used to convert digital (i.e. discrete) data from the time domain into the frequency domain for quantisation and storage. Its name originates as it uses cosine functions to be implemented.

DST (Discrete Sine Transform) - Mostly same as DCT, but uses a sine function to implement it.

Quantisation - This is the process where by the most of the "lossy" compression is performed as transformed values are "rounded" to a close quantisation level.

Subband - This is a single band in the frequency domain that contains a set range of the images frequency components within it.

SBDCT - This is the process where by an image is split up into subbands using either DCT or DST so that the collective bands represent the DCT of the entire image.

Subband Decomposition - This is the process by which an image is subdivided into different frequency bands.

Chapter I : Introduction

Background

With the availability of more powerful personal computers and high speed Digital Signal Processing (DSP) systems to the public, digital motion video is fast becoming the next form of communications method to be. Communications employing motion video have a number of uses, for example, video conferencing, video-phones, remote sensing equipment, and instantly available computer based video archives. However, as with most cases, the advantages of video communications are also accompanied by a major disadvantage in that, the transmission of "raw" motion video requires a large bandwidth. This mainly arises from the sampling process of converting analog video to digital, for the obvious advantages that digital systems have to offer.

Even though today's technology offers channels capable of large bandwidths (e.g. optical fibers), in most cases this bandwidth is not available everywhere, due to factors such as overuse of a channel by multiple sources or modem links etc. For this reason the "raw" video needs to be encoded in such a way as to minimise the bandwidth required, or provide a low bit-rate, while maintaining an acceptable picture quality to recognise the information being transmitted.

As a result of this, currently there are quite a number of different motion video compression and decompression schemes available for implementation. One of the first motion video compression (encoding) schemes used was the MPEG standard, which was derived from the JPEG standard, a single image compression scheme. Together with this MPEG I, MPEG III, Microsoft's Motion Video (AVI), Macintosh's Quick-time (MOV),

Recommendation H.261 by ITU-TS (International Telecommunications Union, Teleconferencing Sector), the new subband schemes and other such encoders, make up most of the currently available digital motion video compression schemes. Generally, schemes such as MPEG I, AVI and MOV tend to be large bandwidth orientated encoding schemes, as they work at around 2 Mbps transmission rates. Due to this they consume very little decode - encode time. The others, however, are designed for video compression with transmission bit-rates of 64 Kbps or below, and hence require more computational time.

With the exception of subband coding schemes virtually all the other schemes tend to function in the same way. They basically sub-divide the image spatially into small sub-blocks, perform some motion compensation or estimation, transform each of these blocks individually, entropy encode the resultant coefficients and transmit the data. The subband approaches however, divide the image into subbands instead of blocks, before entropy encoding and transmission.

Therefore most of the codecs (coder - decoder) mention above, tend to suffer from a 'blocking' effect, depending on how much quantisation is used for each block. With the subband approach, however, this is not present, as the entire image is transformed or filtered and quantised at once so that the entire image loses detail collectively instead of the each block. This is the major advantage of the subband coding that is useful if it is implemented.

Currently, there exists a number of different subband coding techniques, all using their different methods of splitting the image into frequency bands. Most of them tend to use the Wavelet transform in the form of Quadrature Mirror Filters (QMF) to perform the subband division. However, another way to split the individual images into subbands has been described by Y.H. Chan and W.C. Siu in their March 1994 IEEE publication,

"An Approach to Subband DCT Image Coding". This technique employs both discrete cosine and sine transforms on the entire image, so as to convert it in entirety into the frequency domain, which is achieved by converting small bands and adding the bands together. This is a more simpler approach than designing QMF's, their associated filter parameters (i.e. FIRs or IIRs) and convoluting these filters with the image, as tend to be done in more conventional subband compression schemes.

The advantage of subband coding combined with the advantage of having an already developed scheme (Recommendation H.261) that can be easily modified to accept SBDCT, lead to the development of this thesis, which plans to explain the design of a digital motion video compression implementation scheme that puts these two together.

The Need For This Scheme

As indicated before, with digital motion video there usually exists a tradeoff between the transmitted bit-rate and the quality of the images within this encoded bit-stream. Since the standard block transform coding schemes tend to suffer from the "blocking" effects, the new trend is, to look towards coding schemes that employ the subband decomposition approach. These schemes tend to provide both low bit-rates and better quality images. Therefore its necessary to look at other schemes that can provide the same benefits as the subband coding scheme while maintaining the ease at which it can be implemented. By combining both the ITU-TS Recommendation H.261 and the SBDCT scheme by Y. H. Chan and W. C. Siu, reference [4] [5], all three issues of, low bit-rate, better picture quality and ease of implementation are addressed.

Related Main Problem Questions

In order to design this implementation scheme there are three basic problem questions that must be dealt with first. They are..

1. What is the existing H.261 Standard and how good is its performance ?
2. What is the process of subband DCT-DST image decomposition ? and
3. Can these two be combined together to develop a scheme that provides both a low bit-rate and a better image quality at low bit-rates than the block transform scheme ?

The answers to these questions will provide the backbone to this implementation scheme and hence this thesis.

Significance of Such a Scheme

Since the H.261 scheme can be already implemented in hardware and software (even thought with some difficulty) all this scheme provides is a modification to it, so that a new scheme can be developed based on subband DCT coding. Therefore, designers of hardware, for example with VLSI, who already can implement most of the H.261 standard modules, only have to make changes to the existing design instead of redesigning the scheme from scratch. Although the hardware and/or software implementation of the scheme goes beyond the scope of this project it plays an important role in the significance of this scheme's development.

Outline of Project with regards to Scheme

The first objective of the project was to review the different block transform compression schemes available for motion video. The second was to implement one such scheme with software using a PC, and the third, to modify it to improve either the bit-rate, the quality or both.

Chapter II : Review of Literature

Block Transform Coding

With the searches made in different journals (e.g. IEEE) and the Internet resource, many documents on various aspects of transform coding orientated motion video and image compression schemes were located. The more prominent being MPEG (Motion Picture Experts Group) 1, 3, 4 standards and the ITU-TS (formerly CCITT) Recommendation H.261 for video compression, while JPEG (Joint Picture Experts Group), being the more commonly used image compression scheme fitting in to this category.

The JPEG standard basically follows a similar top level algorithm, that involves firstly the sub-division of the picture into blocks after which each block is DCT transformed, quantised, variable length coded and transmitted or stored. The MPEG and H.261 standards basically add a further motion compensation and estimation stage to the JPEG oriented standards. This extra stage basically kept track of the previous frame and used the difference or a motion compensated difference between the previous frame and the current frame to perform the transformation on.

The MPEG-1 standards was designed for bit-rates of approximately 2 Mbps and hence can be used for full motion video such as movies. On the other hand the initial H.261 recommendation was designed for bit-rates of 64 Kbps or below as this was intended to be used for teleconferencing purposes, which depended on low motion video. For this reason there are many hardware cards currently available that can encode and

decode movies, documentaries using the MPEG standard. In comparison, the MPEG 3 and 4 standards were newly designed to be able to match the bit-rate of the H.261 standard (these were developed recently and were not as easy to come by as the H.261 recommendation). The only disadvantage with this coding technique is that under harsh quantisation conditions the images suffered from "blocking" effects that distorted it.

Subband Coding

Since the concept of using subband coding schemes for image and motion video compression is more advantageous and newer, its is quite well elaborated on in the literature. Subband coding was first introduced in 1976 by R. E. Crochiere, S. A. Webber, and J. L. Flanagan for coding of speech signals. From this beginning it was adopted to be used in image coding by many different researchers. Hence subband coding schemes can be implemented in a number of ways. The more theoretical, but original way was to use the Wavelet transform and a suitable mother wavelet on the signal data in question. This transform, which behaves similar to the human visual system's image frequency interpretation, eliminates the use of a time window for transforming data, hence making it a better transform than the Discrete Fourier Transform for time-frequency signal analysis. This is mainly due to the wavelet transform being a finite transform, unlike the sinusoidal based Fourier transforms. For more information please read an excellent article written by O. Rioul and M. Vetterly, Oliver Rioul et al. 1991 [19] on wavelets.

This transform had the effect of splitting the image into a number of frequency subbands (hence the name) that could be encoded by taking to account the characteristics of that particular subband when encoding. As a result most compression techniques that employ subband coding, convert the entire image into frequency

subbands, quantise each of the subbands according to the characteristics of that subband and then entropy encodes these bands.

Despite, its origin through wavelet transforms, the subband coding based schemes are more commonly realised by the use of more simpler high and low pass 2-dimentional filters. The filters used are usually two dimensional Quadrature Mirror Filters (QMF). QMFs are used as they tend to cancel the aliasing effect in the encode/decode process. The aliasing effect comes to being as, since the filters produce two sets of data the same size of the image, they have to be down sampled by two so that they come back to the image size. Therefore, by using QMFs the aliasing present in the encode process is canceled out by the up-sampling done in the decode process. The subband coding of images using one dimensional QMFs to emulate a two dimensional QMFs is discussed by J. W. Woods and S. D. ONeil, J. W. Woods et al 1985 [23]. The basic QMF subband division and reconstruction process can be seen in Figure 1. While Figure 2 shows the resulted subband separation within the image size. Figures obtained from "Subband Coding Of Images", J. W. Woods and S. D. O'Neil.

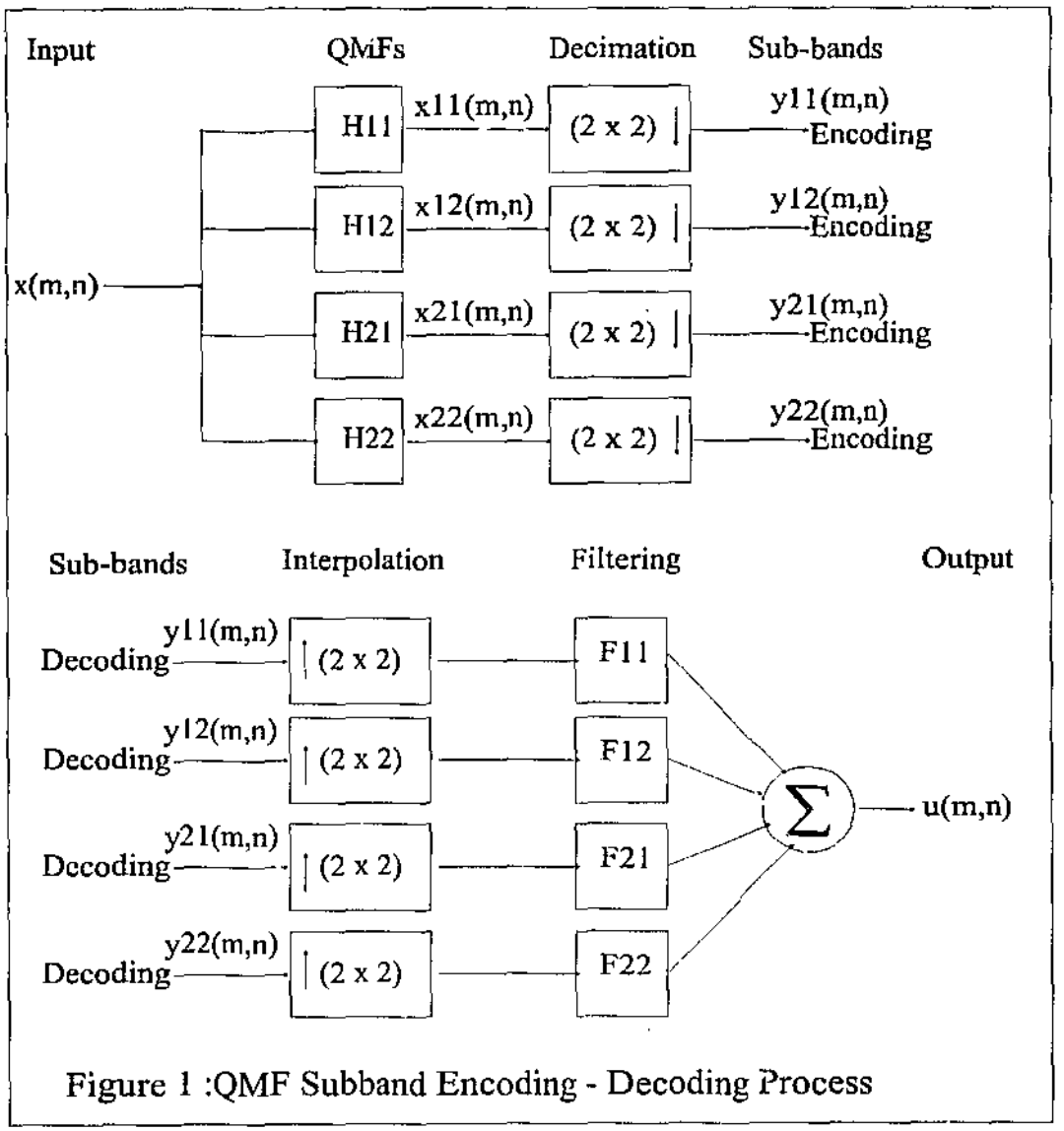


Figure 1 :QMF Subband Encoding - Decoding Process

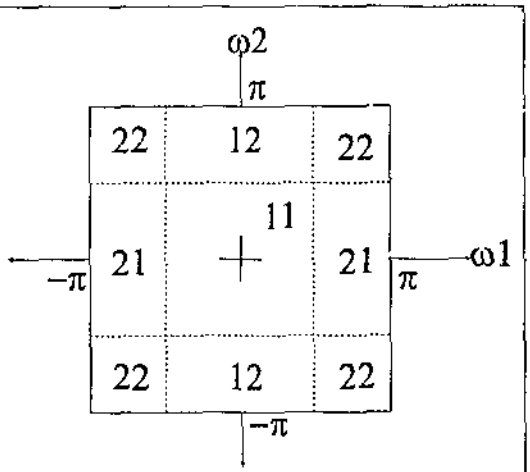


Figure 2 : 4 Band Image Split.

Aside from the previous two implementations of subband coding schemes, two other researchers have developed a scheme whereby an entire image can be discrete cosine transformed by splitting it into subbands . In 1993, Y. H. Chan and W. C. Siu, Y. H. Chan et al 1993 [4], introduced an approach, which considered the DCT for the whole frame. The DCT provided the conversion of the image into the frequency domain, which could then be quantised and stored. However, they claim that a direct evaluation of the DCT on the image would require too much computational power, therefore they developed a system where the overall DCT could be broken down into smaller DCTs and DSTs. This, they claimed, although not being the same as conventional subband splitting, was a form of frequency segmenting.

Basically this scheme recursively splits the image into four subbands each time until the required block size is attained. Once this is completed, by simply "fitting" each transformed block next to its adjacent blocks, the DCT of the entire image is obtained. The more recursive levels used the less computational power is required for the cosine and sine multiplication loops. However, they claimed that, splitting the image into too smaller blocks will increase the reconstruction complexity of the DCT and hence a compromise between the two is required. This type of scheme is claimed to very similar to the work done on using Laplacian pyramidal structures to decompose the image (Originally done by P. J. Burt and E. H. Adelson) . Figure 3 Shows the pyramidal structure employed by this subband coding scheme.

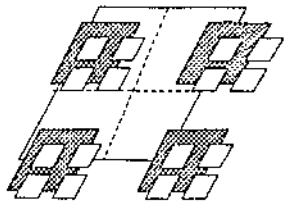


Figure 3 : Pyramidal Structure

The advantages of this scheme to this implementation scheme are two fold.,

1. Since QMFs are not used, complex filter design is not necessary and as a result it can be easily adopted to fit into the Recommendation H.261.
2. 2. It eliminates the "blocking" effect inherent to the block transform coding schemes but still uses DCTs.

An interesting, publication from K. N. Ngan and W. L. Chooi 1994 [16] explained how a 3-dimentional subband approach can be used to not only to segment images spatially but also segment it in blocks in terms of multiple frames. That is to use QMFs on the temporal domain. Since each subsequent frame carries information with a little difference from the current image, this temporal difference can be filtered like an image. The advantage of this scheme is that it eliminates the process of time consuming motion searches between frames for vectors. However, since this scheme requires a complete restructure of the coding mechanism it is not used in this scheme, but viewed as an improvement that could be made to the scheme.

Chapter III : The ITU-TS Recommendation H.261

Introduction

In 1990, the International Telecommunications Union - Teleconferencing Sector finalised a standard for digital motion video encoding - decoding and named it Recommendation H.261. The standard provided a publicly available video compression - decompression scheme that could be used in applications such as the video-phone, video conferencing equipment. The recommendation was made to replace the existing standard named Recommendation H.120, so as to be able to provide coded bit-rates of at least 64Kbps or below. This rate was a primary target as the ISDN networks provided channels such as B, H_0 and H_{11}/H_{12} which operated at around 64 Kbps or higher. A device employing this technique would contain the same basic functional blocks depicted in Figure 4.

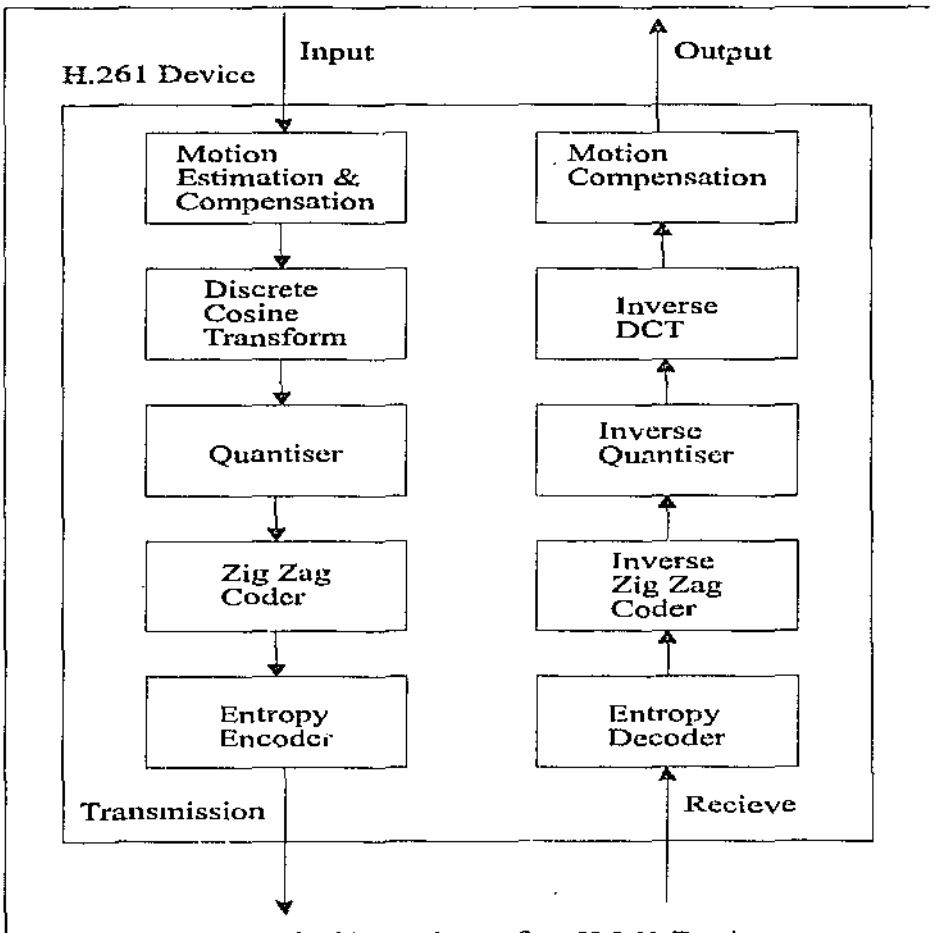


Figure 4 : Typical Functions of an H.261 Device

The Overall Compression - Decompression Process.

The overall Compression and Decompression schemes can be seen in Figures 5 and 6 respectively.

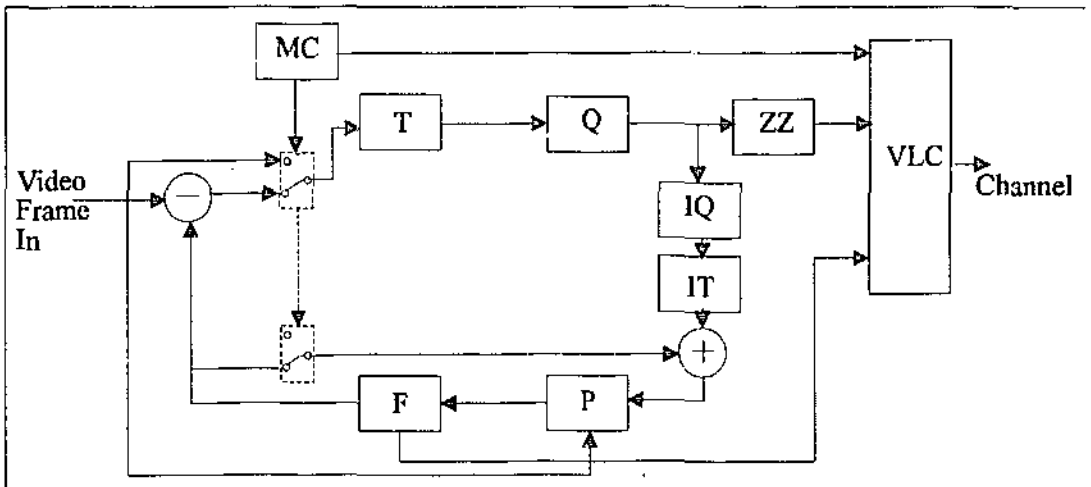


Figure 5 : H.261 Overall Encoding Process

Where,

T = DCT Transform
IT = Inverse DCT Transform
MC = Motion Compensation
P = Picture Memory
VLC = Variable Length Encoder
Q = Quantiser
IQ = Inverse Quantiser
ZZ = ZigZag Encoder
F = Loop Filter

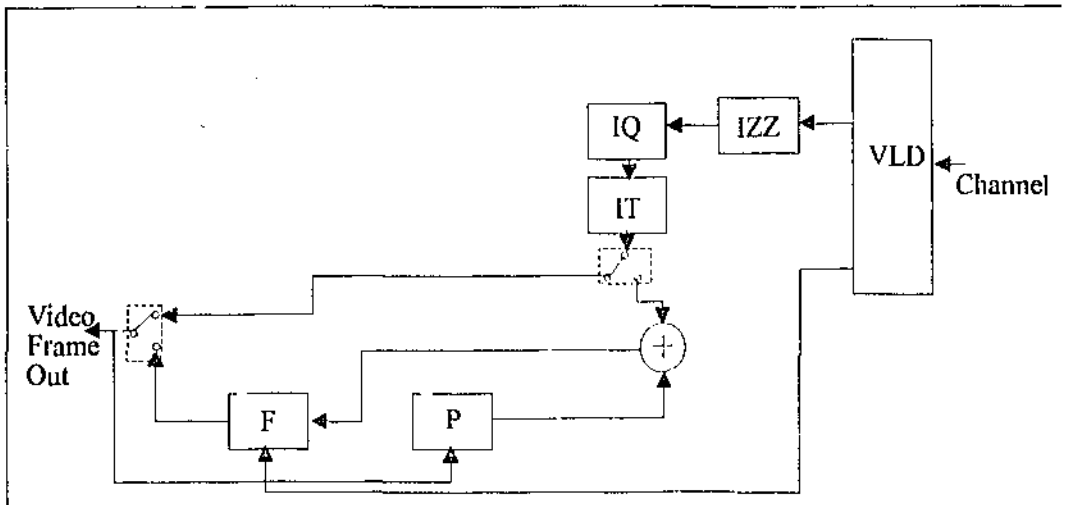


Figure 6 : H.261 Overall Decoding Process

Where,

VLD = Variable Length Decoder
IT = Inverse DCT Transform
F = Loop Filter
IZZ = Inverse ZigZag Encoder
IQ = Inverse Quantiser
P = Picture Memory

The encoding scheme firstly checks to see for the existence of a previous frame, if it does not find one, it will transform, quantise, zigzag encode and variable length encode the frame. Then it will inverse quantise and inverse transform this frame before storing it in temporary memory. If a previous frame is encountered then the previous

frame is subtracted from the current frame according to the scheme implemented in the motion compensation stage. This difference is then transformed, quantised, zigzag encoded and entropy encoded. This time in addition to the inverse quantisation and transform it will also add the previous frame to the current difference, then store it in temporary memory.

The decoding process on the other hand firstly decodes the variable length code, inverse zigzags the image before its inverse quantised and transformed. Now if the previous frame did not exist, this frame will be put straight out, else the previous frame's content will be added to the current image, before filtering and releasing as a frame. This released frame is then copied into the internal storage to use with the next frame. The filter process is to combat some of the effects of the windowing done, due to the block segmentation of the frame.

The Image Format

The H.261 scheme is designed around the manipulation of two frame types, the PAL CIF and QCIF (Quarter CIF). The NTSC frame type can also be supported, however it needs to be indicated. These frame types indicate the size of the image as can be seen in Table 1. Its worth noting that only QCIF sequences are required to conform to the 64Kbps bit-rate.

Type	Y-HIG	Y-WID	U-HIG	U-WID	V-HIG	V-WID
CIF	352	288	176	144	176	144
QCIF	176	144	88	72	88	72
NTSC	352	240	176	120	176	120

Table 1: Image Formats

Each frame is divided into three components in a similar fashion to that used by standard television signals. That is, they contain a luminance frame (Y - Frame) and two chrominance frames (U and V Frames). This colour scheme is chosen as the when using the standard RGB scheme the green is easily distorted. From Table 1 it is clear that the Y frame in all cases is four times as large as the U or V frames. This is because one U or V pixel is used for four luminance pixels. Therefore, pixels are sampled as shown in Figure 7.

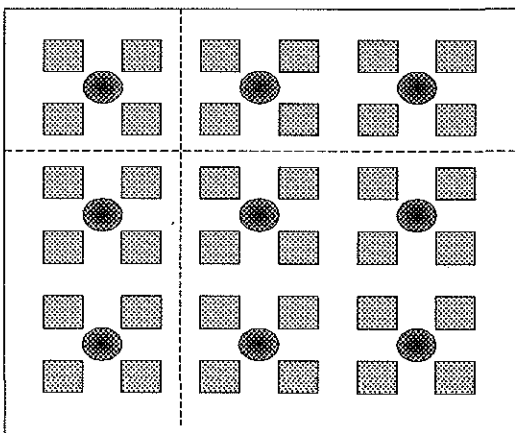


Figure 7 :
H.261 Pixel Arrangement
Luminance Pixel
Chrominance Pixel

The alignment of the pixels in this fashion is the first step towards the whole image compression process. Usually, in a RGB representation of an image, 24 bits are used to represent an image size of a Y frame. However, using this scheme, on average only 12 bits can be used to represent a pixel on the same image. This is possible because the chrominance varies only slightly from one luminance pixel to another (the reason why black & while television images are more high definition than their colour counterparts). The conversion between RGB ←→ YUV, can be made if the following matrices are computed.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

The input Y frame is a sequence of 8 bit values signifying the levels of the luminance through out the picture, while the U and V frames are sequences of values from -128 to 127, level shifted by adding 128 to the value.

Image Subdivision

As mentioned previously, like all block transforms compression techniques the H.261 scheme divides the image into a number of sub-images. The smallest of sub-division is a block with dimensions 8 x 8 pixels. Four of these blocks are combined to form a super-block, for the Y frame only. One super-block and two corresponding 8 x 8 blocks from the U and V frames go to make up a macro-block. 33 macro-blocks set in three rows of 11 macro-blocks, go to construct a Group Of Blocks (GOB) structure. A CIF image is divided into six rows of two GOBs, while a QCIF image is divided into just three GOBs. The NTSC frame is stored as a CIF with a special indication, that sets the limit to five rows of two GOBs. See figure 8 for the image sub-division. Higher, order divisions are made so that only parts of the image can be transmitted, instead of transmitting the entire image.

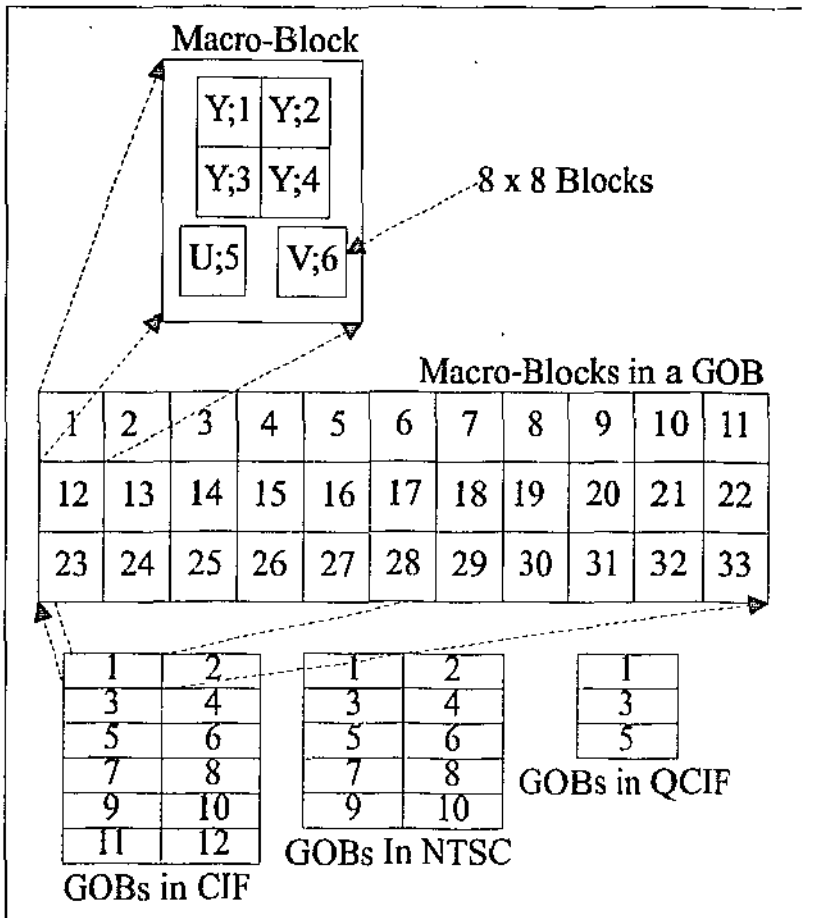


Figure 8 : Frame Sub-Division

Motion Compensation and Estimation.

This stage of the encoder deals with two basic phenomena,

1. The lack of temporal change and
2. The effect of panning or slow moving objects.

It is quite well known that most video sequences have redundant information from one frame to the next. This is because there only seems to be a center of interest that changes intensities and colours regularly, while backgrounds and other still objects remain in the same place for quite some time. Therefore, if only the moving parts are transmitted, the others can be extracted form the

previous frame. To add to this if the difference between the current and previous frames are taken then the change in values in a block, will be smaller, on average than if a "raw" information is transmitted. This is useful as this provides more zeros after a transform. This is called Motion Compensation.

If the image pans or a large block moves across then the simple difference technique is not enough as the difference between blocks in the same position, in the current and previous frames may be greater than the "raw" image blocks values combined. Two things can be done here

1. The frame can be coded as a "raw" sub-image. Or
2. 2. A search can be made to see if that block moved anywhere and if so use the difference at that position instead. This is termed Motion Estimation.

The Recommendation H.261 has the ability to use any or both of these, and usually operates by doing a -15 to 15 (30) pixel motion search in both the horizontal and vertical direction on the Y frame. In doing so, it searches through all 30 x 30 combinations comparing the absolute pixel to pixel difference between the current block in question and the surrounding blocks in the previous frame, to see which gives the least difference. Once the proper block is found its variance is compared with that of the variance found in the original "raw" block. If it is smaller, then motion compensation is used on the motion estimated macro-block (inter or MC + inter macro-block is generated), else the macro-block is coded as an intra ("raw") macro-block. If motion compensation is used on a motion estimated macro-block, then a two value motion vector is kept with that macro-block. Since its only the Y - frame that under goes motion compensation and estimation any vectors are divided by two for the U and V frames.

The actual H.261 doesn't specify a method for the calculation of motion compensation or estimation explicitly, however a recognized implementation of this scheme, by Andy C. Hung, operates as described above. On the whole the process of motion compensation and estimation is a very slow process that needs improvement.

Transform (DCT and IDCT)

The Recommendation H.261 defines a strict Discrete Cosine transform and an Inverse Discrete Cosine Transform that is to be applied to all 8 x 8 blocks to be coded. The transforms are...

DCT

$$F(u,v) = (1/4)C(u)C(v)\sum_{i=0}^{7}\sum_{j=0}^{7} f(i,j)\cos((2i+1)u\pi/16)\cos((2j+1)v\pi/16)$$

IDCT

$$f(i,j) = (1/4)\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos((2i+1)u\pi/16)\cos((2j+1)v\pi/16)$$

where

$$C(g) = \begin{cases} 1/\sqrt{2}\ ;\ g = 0; \\ 1\ ;\ \text{otherwise} \end{cases}$$

As can be seen, the forward and inverse transforms are very similar, as can be found when employing orthogonal transforms, quite unlike the DFT. The transform is necessary as it converts a particular block into the frequency domain, with the top-left hand value indicating the low frequency or DC component and the bottom-right hand block indicating the most high frequency present. This is equivalent to a 2-dimentional representation of the frequencies present in the image block as displayed in Figure 9. This figure doesn't show it but the wave-form can fluctuate above and below the frequency axis, as at a particular frequency the level can be positive or negative.

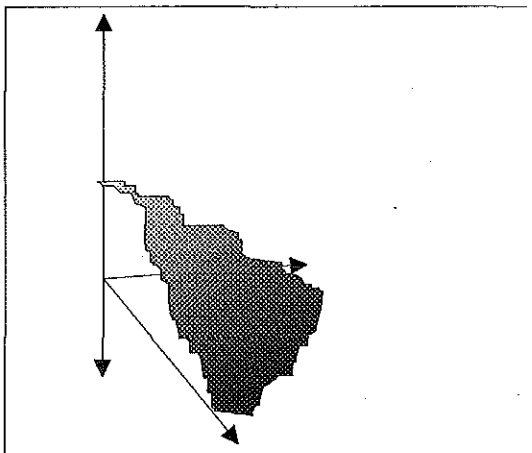


Figure 9
2D DCT Frequency Representation.

An example provided by A. C. Hung [9] demonstrates the result of a DCT on a simple image (pizza - figure 10) and its transform Table 2.

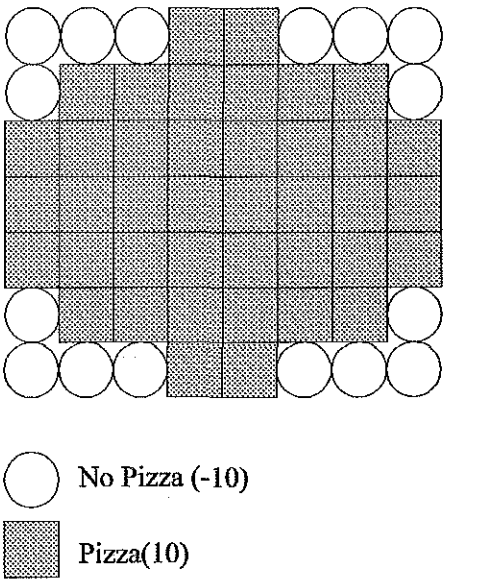


Figure 10 : Simple Pizza Image

Table 2: Transformed Pizza.

40	0	-26	0	0	0	-11	0
0	0	0	0	0	0	0	0
-45	0	-24	0	8	0	-10	0
0	0	0	0	0	0	0	0
-20	0	0	0	20	0	0	0
0	0	0	0	0	0	0	0
-3	0	10	0	18	0	4	0
0	0	0	0	0	0	0	0

As can be seen in the example the image is transformed to a pattern with a lot more zeros than the original and hence can be run length encoded more efficiently.

Quantisation and Inverse Quantisation

Once the DCT of a block is attained, this block's coefficients can then be quantised so as to produce more zeros. Quantising in the frequency domain doesn't seem to effect the image as much as quantising the image in the spatial domain and hence its applied this way. The H.261 recommendation employs the following quantisation and inverse quantisation schemes.

If Quant is odd then $F_Q(u,v) = F(u,v) / (2Quant)$

if Quant is even then $F_Q(u,v) = (F(u,v) \pm 1) / (2Quant)$, where the $\pm$ is positive if for F(u,v) > 0, and negative if F(u,v) < 0.

For the inverse case..

if Quant is odd then $F(u,v) = (F_Q(u,v) \pm 1)(Quant)$,

if Quant is even then $F(u,v) = (F_Q(u,v) \pm 1)(Quant) \mp 1$, where for $F_Q(u,v) > 0$, $\pm$ is positive and $\mp$ is negative. While for $F_Q(u,v) < 0$ $\pm$ is negative and $\mp$ is positive.

The Quantisation process is the heart of the "lossy" compression scheme employed by the recommendation H.261, as the run length and variable length coding scheme's success depends on the adequate quantisation of the transform coefficients. The recommendation suggests that a Quantisation or Quant between 1 and 31 is used and hence allocates six bits for the quantisation storage (In terms of MQUANT - Macro-block quantisation or GQUANT - GOB quantisation).

ZigZag Encoding and Decoding

Once the coefficients have been quantised they need to be converted from a 2-dimentional structure to a 1-dimentional structure, so that the zeros can be run length encoded. Basically, the H.261 scheme does this by performing zigzag process from position (1,0) to (0,1) to (2,0) to (1,1)..(7,7). See Figure 11. This is only done for the AC coefficients. However, for inter and MC + inter macro-blocks the DC is considered as an AC coefficient and can be treated similarly. Intra macro-blocks treat the DC separately.

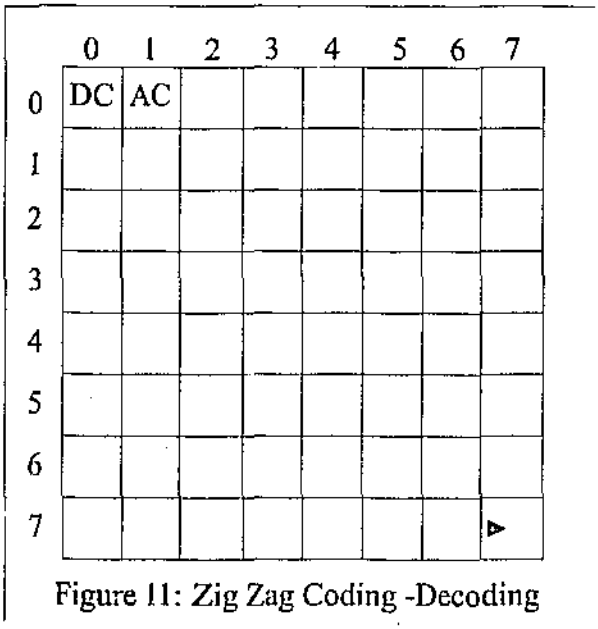


Figure 11: Zig Zag Coding -Decoding

Entropy (VLC - Huffman) Coding

It is this part that converts the operated macro-block structure into the fully defined H.261 bit-stream data for transmitting. Since the actual VLC codes are fully defined in the ITU-TS Recommendation H.261, the tables are not repeated here, but the general structure of the bit-stream is looked at instead.

The first code sent is the PSC or Picture Start Code, which is really a GSC (GOB Start Code) with the GOB number set to zero. This always signifies the start of a frame. The next few bits define the picture characteristics, i.e. CIF, QCIF, NTSC etc. Its worth noting that the initial H.261 specification didn't handle NTSC files but it was added by enabling the extra field to indicate the presence of a NTSC frame.

The next code transmitted or received is the GSC (Group of blocks Start Code), followed by the GOB number and GQUANT (The Group of blocks Quantiser as the H.261 scheme allows a global quantiser for a group of blocks as well as for each macro, MQUANT). Next a macro-block address, found within this GOB is sent (or bit stuffing for EOF), which is followed by the type of macro-block. There are 10 different types of macro-blocks.

1. Intra - This is a macro-block that contains coefficients that do not depend on the previous frame for any motion compensation.
2. Intra + MQUANT - This is the same as 1., but has its own block quantiser present in the next few bits. (used for adaptive quantising)
3. Inter - A macro-block in this form contains only motion compensated data and requires a previous frame. However, it contains no motion vectors (for example background type blocks tend to be in this type). It also contains a VLC CBP code that indicates which of the 8 x 8 blocks in this particular macro-block are present. (This is required as the coefficient VLC encoder is not capable of indicating blocks that don't have non zero coefficients, that can be caused by the quantisation process). The CBP is calculated by allocating six bits, one for each of the six blocks in a macro-block. Therefore 32 = block 1 only while 33 = block 1 and block 6.
4. Inter + MQUANT - Same as 3., but the next field is a quantiser for all the coefficients in this macro-block, followed by CBP.

5. Inter + MC - This type of macro-block, is followed by VLC encoded horizontal and vertical motion vectors, but no coefficients. Unless, the previous macro-blocks were 1, 12, 23, not MC or the current macro-block address shows a difference of greater than one from the last macro-block, the motion vectors are calculated by adding the previous macro-block's vectors to the current motion vectors. If this is not the case then the motion vectors are exactly what they are decoded as. This sort of macro-block just copies, the data, pointed to by the motion vector, from the previous frame.
6. Inter + MC + Coff. - This is similar to 6., but contains the motion vectors followed by a CBP and then the coefficients. The motion vectors and CBP are calculated as explained before. However the image is not just copied from the motion vector, but summed with the contents pointed to by the current motion vector on the previous frame.
7. Inter + MC + MQUANT + Coff. - This is similar to 6., but has its own quantiser following the macro-block type indicator.

8., 9., 10, - These are identical to 5., 6., and 7. Except that the resultant image goes through a loop filter to cancel the effects caused by windowing etc.

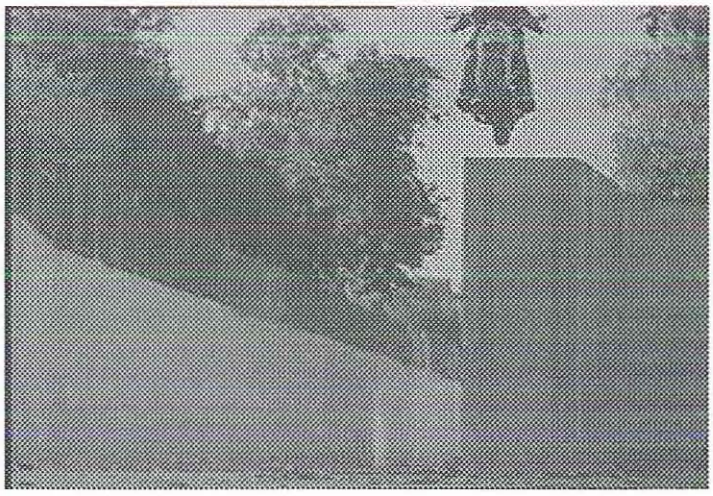
Finally the coefficients are encoded. For intra macro-blocks the first level is transmitted as a signed integer. For inter, MC and the other coefficients of an intra macro-block, the number of zeros before a non zero value is counted. This is termed the Run, while the Level is the next nonzero value. Each run-level combination is looked up in a coefficients VLC table and its corresponding bit patterns are transmitted. This is done until its there are no more non zero values, upon which an End Of Block is transmitted. Any zeros in a block after the last not zero level are discarded. This is the reason why more zeros are made to be together by all three functional components, the transform, quantisation and zigzag encoding process.

Advantages of H.261

The main advantage of the H.261 digital video compression recommendation is that it allows for bit-rates 64Kbps and below, although depending on the frame size and rate used.

Disadvantages of H.261

Like all block transform coding systems, the major disadvantage with the H.261 recommendation is that it is susceptible to the "blocking" effect, caused by harsh quantisation. Since harsh quantisation originates from an effort made to make more zeros the inverse transform tends to get stuck transforming a few levels with many zeros, this leads to the "smuggy - blocking" effect on the image. Figure 12 shows the same image quantised with GQUANT = 8 and GQUANT = 31. Its is clear that the latter image is more blocky and smudged. (Image - bike)



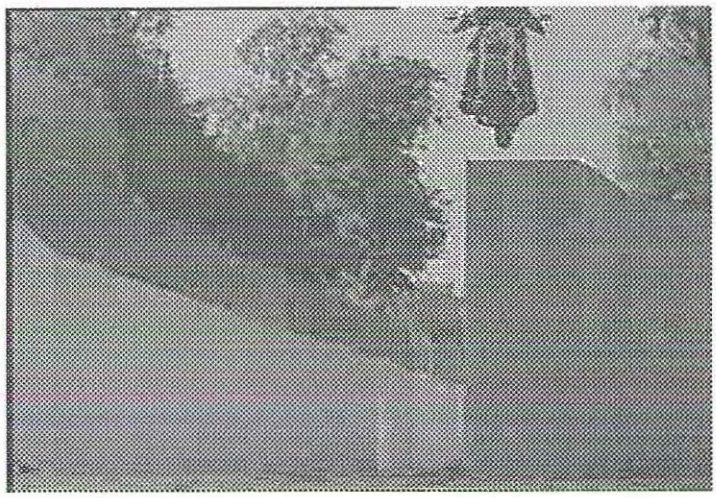


Figure 12 : Quantisation Effects

H.261 Compression Performance

Table 3 is a indication of the compression that can be achieved by employing the H.261 Scheme.

File Name	Picture Format	# Of Frames	Comp. Size	Raw Size	Ratio
bike.p64	NTSC	150	395,329	19,008,000	1:48
short.p64	NTSC	7	52,404	1,599,064	1:30
stennis.p64	NTSC	150	871,662	19,008,000	1:22
sflowg.p64	NTSC	150	3,190,863	19,008,000	1:6

Table 3: H.261 Compression

Chapter IV : Subband DCT Image Coding Approach

Introduction

In March 1994 Y. H. Chan and W. C. Siu, Y. H. Chan 1994 [4] wrote an article named "An approach to Subband DCT Image Coding", which explained a different way of subband DCT coding an image, to give better results than a standard block transformed image coding system. The better result was mainly the elimination of the previously mentioned "blocking" effect. They claimed that most block oriented transforms used small blocks on which to perform the DCT as this reduced the computational effort. However, smaller block sizes, especially with heavy quantisation, tended to introduced the blocking effect. Their next suggestion was to perform a DCT on the whole image at once, as this would eradicate the existence this blocking effect, which was a result of performing DCTs on the individual sub-images.

They however, acknowledged that to perform this they would need a large computational effort. Therefore, they made a closer examination of the existing 1-dimentional DCT transform and found that it could be represented by two different transforms a DCT and a discrete sine transform. The resultant two transforms also only used half the number of samples to compute the original DCT. They also discovered that this process could be recursively used until the block sizes for the transform were small enough to be effectively computed. Once these smaller transforms were computed they could be combined together with adjacent transforms to completely represent the original DCT. This splitting structure into DCTs and DSTs, they claimed it to be a subband

frequency division process, even though it did not use the conventional Quadrature Mirror Filters or related techniques. Hence they termed it SBDCT.

The Subdivision Scheme

Once the 1-dimentional, conditions were established they extended it into the 2-dimentional domain for images. The result was a number of equations that could be used recursively to perform a 2-dimentional DCT on a whole image by splitting the image into subbands. A summery of their discovery in terms of final equations is now displayed.

2-D Discrete Cosine - Cosine Transform - This is the original transform to be performed on the entire image.

Forward Transform (CCT)

$$X_{cc}(m,n) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x(i,j)\cos(\frac{m\pi(2i+1)}{2N})\cos(\frac{n\pi(2j+1)}{2N})$$

$$for\ m,n = 0,1,\ldots\ldots,N-1$$

Inverse Transform (ICCT)

$$x(i,j) = \sum_{m=0}^{N-1}\sum_{n=0}^{N-1} \delta(m)\delta(n)X_{cc}(m,n)\cos(\frac{m\pi(2i+1)}{2N})\cos(\frac{n\pi(2j+1)}{2N})$$

$$for\ i,j = 0,1,\ldots\ldots,N-1$$

2-D Discrete Cosine - Sine Transform

Forward Transform (CST)

$$X_{cs}(m,n) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x(i,j)\cos(\frac{m\pi(2i+1)}{2N})\sin(\frac{n\pi(2j+1)}{2N})$$

$$for\ m = 0,1,\ldots,N-1; n = 1,2,\ldots,N$$

Inverse Transform (ICST)

$$x(i,j) = \sum_{m=0}^{N-1}\sum_{n=1}^{N} \delta(m)\delta(n)X_{cs}(m,n)\cos(\frac{m\pi(2i+1)}{2N})\sin(\frac{n\pi(2j+1)}{2N})$$

$$for\ i,j = 0,1,\ldots,N-1;$$

2-D Discrete Sine - Cosine Transform

Forward Transform (SCT)

$$X_{sc}(m,n) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x(i,j)\sin(\frac{m\pi(2i+1)}{2N})\cos(\frac{n\pi(2j+1)}{2N})$$

$for\ m = 1,2,\ldots,N; n = 0,1,\ldots,N$

Inverse Transform (ISCT)

$$x(i,j) = \sum_{m=1}^{N}\sum_{n=0}^{N-1} \delta(m)\delta(n)X_{sc}(m,n)\sin(\frac{m\pi(2i+1)}{2N})\cos(\frac{n\pi(2j+1)}{2N})$$

$for\ i,j = 0,1,\ldots,N-1;$

2-D Discrete Sine - Sine Transform

Forward Transform (SST)

$$X_{ss}(m,n) = \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x(i,j)\sin(\frac{m\pi(2i+1)}{2N})\sin(\frac{n\pi(2j+1)}{2N})$$

$for\ m,n = 1,2,\ldots,N$

Inverse Transform (ISST)

$$x(i,j) = \sum_{m=1}^{N}\sum_{n=1}^{N} \delta(m)\delta(n)X_{ss}(m,n)\sin(\frac{m\pi(2i+1)}{2N})\cos(\frac{n\pi(2j+1)}{2N})$$

$for\ i,j = 0,1,\ldots,N-1;$

where, $\delta(k) = \frac{1}{N}\ if\ k = 0 \text{ or } N;\ = \frac{2}{N}$ else

To achieve the subband splitting each of these function have to be able to be represented in terms of smaller transforms. The equations to follow can be used on both the forward and the inverse transforms.

$$X_{cc}(m,n) = CC(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_0(i,j)\cos(\frac{m\pi(2i+1)}{N})\cos(\frac{n\pi(2j+1)}{N})$$

$$+ CS(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_1(i,j)\cos(\frac{m\pi(2i+1)}{N})\sin(\frac{n\pi(2j+1)}{N})$$

$$+ SC(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_2(i,j)\sin(\frac{m\pi(2i+1)}{N})\cos(\frac{n\pi(2j+1)}{N})$$

$$+ SS(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_3(i,j)\sin(\frac{m\pi(2i+1)}{N})\sin(\frac{n\pi(2j+1)}{N})$$

$for\ m,n = 0,1,\ldots,N-1$

$$X_{cc}(m,n) = CC(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_0(i,j)\cos(\frac{m\pi(2i+1)}{N})\sin(\frac{n\pi(2j+1)}{N})$$
$$- CS(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_1(i,j)\cos(\frac{m\pi(2i+1)}{N})\cos(\frac{n\pi(2j+1)}{N})$$
$$+ SC(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_2(i,j)\sin(\frac{m\pi(2i+1)}{N})\sin(\frac{n\pi(2j+1)}{N})$$
$$- SS(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_3(i,j)\sin(\frac{m\pi(2i+1)}{N})\cos(\frac{n\pi(2j+1)}{N})$$
$$for\ m = 0,1,\ldots,N-1; n = 1,2,\ldots,N;$$

$$X_{sc}(m,n) = CC(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_0(i,j)\sin(\frac{m\pi(2i+1)}{N})\cos(\frac{n\pi(2j+1)}{N})$$
$$+ CS(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_1(i,j)\sin(\frac{m\pi(2i+1)}{N})\sin(\frac{n\pi(2j+1)}{N})$$
$$- SC(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_2(i,j)\cos(\frac{m\pi(2i+1)}{N})\cos(\frac{n\pi(2j+1)}{N})$$
$$- SS(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_3(i,j)\cos(\frac{m\pi(2i+1)}{N})\sin(\frac{n\pi(2j+1)}{N})$$
$$for\ m = 1,2,\ldots,N; n = 0,1,\ldots,N-1;$$

$$X_{ss}(m,n) = CC(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_0(i,j)\sin(\frac{m\pi(2i+1)}{N})\sin(\frac{n\pi(2j+1)}{N})$$
$$- CS(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_1(i,j)\sin(\frac{m\pi(2i+1)}{N})\cos(\frac{n\pi(2j+1)}{N})$$
$$- SC(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_2(i,j)\cos(\frac{m\pi(2i+1)}{N})\sin(\frac{n\pi(2j+1)}{N})$$
$$+ SS(m,n)\sum_{i=0}^{N/2-1}\sum_{j=0}^{N/2-1} y_3(i,j)\cos(\frac{m\pi(2i+1)}{N})\cos(\frac{n\pi(2j+1)}{N})$$
$$for\ m,n = 1,2,\ldots,N$$

where

$$y_0(i,j) = x(2i,2j) + x(i,2j+1) \div x(2i+1,2j) + x(2i+1,2j+1)$$
$$y_1(i,j) = x(2i,2j) - x(i,2j+1) + x(2i+1,2j) - x(2i+1,2j+1)$$
$$y_2(i,j) = x(2i,2j) + x(i,2j+1) - x(2i+1,2j) - x(2i+1,2j+1)$$
$$y_3(i,j) = x(2i,2j) - x(i,2j+1) - x(2i+1,2j) + x(2i+1,2j+1)$$
$$for\ i,j = 0,1,\ldots.N/2-1$$

and

$$CC(m,n) = \cos(\frac{m\pi}{2N})\cos(\frac{n\pi}{2N})$$
$$CS(m,n) = \cos(\frac{m\pi}{2N})\sin(\frac{n\pi}{2N})$$
$$SC(m,n) = \sin(\frac{m\pi}{2N})\cos(\frac{n\pi}{2N})$$
$$SS(m,n) = \sin(\frac{m\pi}{2N})\sin(\frac{n\pi}{2N})$$
$$for\ m,n = 0,1,\ldots.N$$

A reason for the authors to claim that this is similar to subband scheme is due to the CC, CS, SC and SS definitions, which are a set of high and low pass filters, with cosine being the low and sine being the high. The authors claim that in the actual implementation process the CC, CS, SC and SS can be replaced by ideal filters for image compression mainly without effecting the resultant image. This reduces the number of cosine (sine) multiplication's that have to be used, as filters composed of just 1's and 0's for pass and stop bands can be used.

The Overall Coding - Decoding process

The coding process initially begins by trying to take the DCT of the entire image. Now since this is computation intensive, using y_0 to y_3 and X_{cc} the image is split up and into four subbands. This step is termed Quartering. Now that there are four transforms that each containing ¼ the of the image's information and since they are still large, each of these subbands can be quartered again, to give 16 subbands. This process can be continued on until block sizes reach 256 x 256 or 16 x 16 etc. Its worth noting

here that the reconstruction tracking is bound to take quite a computational effort if the block sizes get too small.

Once the encoded image, which is really the DCT of the full image, is at hand, a inverse DCT can be applied to the entire image, since these are orthogonal transforms, the same Quartering process can be implemented on the inverse DCT quantisation. Therefore, four inverse transforms will come into presence, each of these can then be further sub-divided to reduce the computational effort.

The main advantages of this scheme include, the elimination of the blocking effect and the inherent low bits per pixel (approx. 0.5 bits per pixel).

Chapter V : The possible usage of SBDCT with the H.261 Recommendation to produce a new coding scheme.

Introduction

This document is based on the design of an digital video encoding implementation scheme that has the features of both, low bit-rates (64-Kbps or below) and better image quality than the ordinary block transform method. Since the recommendation H.261 provides the bandwidth compression using DCT blocks in a sub-image, and since the SBDCT image coding scheme provides the better quality images, this chapter will describe the changes that would need to be done to the H.261 scheme in order to accommodate the subband DCT coding system.

The Overall Compression - Decompression Process

The overall encoding scheme can be seen in Figure 13. It shows two differences from the original H.261 scheme, 1. The DCT transform is replaced by a unit that enlarges the frame, performs a SBDCT and a spliter for the resultant DCT. 2. The Inverse transform stage, that is now replaced a unit that combines the image, performs an Inverse SBDCT and deflates the size of the image. The necessity of there are described in the next section.

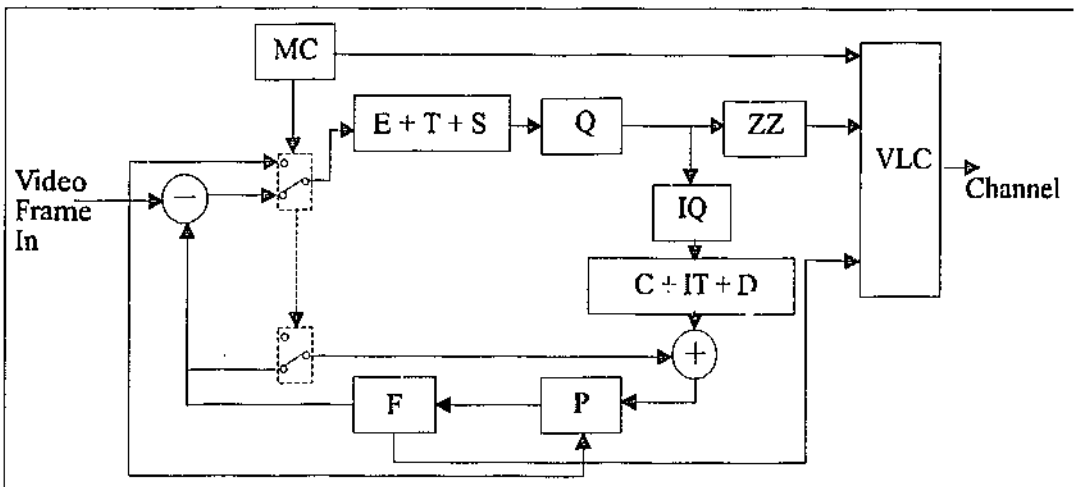


Figure 13 : The Modified Overall Encoding Process

Where,

T = SBDCT Transform
IT = Inverse SBDCT Transform
MC = Motion Compensation
P = Picture Memory
VLC = Variable Length Encoder
D = Deflater
E = Enlarger
Q = Quantiser
IQ = Inverse Quantiser
ZZ = ZigZag Encoder
F = Loop Filter
C = Combiner
S = Splitter

Figure 14 shows the overall diagram for the decoder, which has the H.261 DCT inverse transform unit replaced by unit that combines the SBDCT image, performs an ISBDCT and deflates the image to original size.

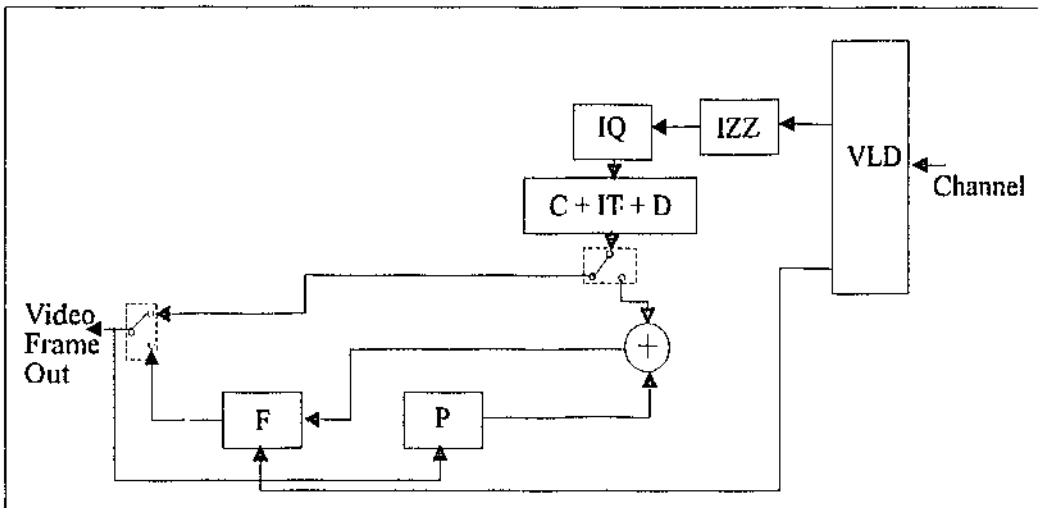


Figure 14 : The Modified Overall Decoding Process

Where,

IT = Inverse SBDCT Transform
P = Picture Memory
IQ = Inverse Quantiser
F = Loop Filter

VLD = Variable Length Decoder C = Combiner
D = Deflater

By, allowing for this SBDCT scheme the original H.261 requires modification in five basic parts. 1. The frame size and hence the division. 2. The transform, 3. The Motion Compensation area, 4. The coding scheme and 5. The Quantising scheme.

The Frame Size

The input, output and previous frame image sizes remain the same as for the H.261 encoder - decoder (i.e. support for NTSC, CIF and QCIF). However, the working frame's size will have to be modified so that the width of the image is the same size as the height. This is because the SBDCT transform works on images that are square (i.e. perfect division by four). Therefore, extra rows are needed and can be set to zero, as this does not seem to effect the overall DCT transform (experience gained while implementing the H.261 scheme). When the image is decoded these rows can be discarded. Table 4 shows the rows wasted for each type of frame.

Type	Y	U	V
CIF	64	32	32
QCIF	32	16	16
NTSC	112	56	56

Table 4 : Wasted Lines

Although these lines are wasted in the image, when transformed these should provide many zeros that will be run-length encoded efficiently.

Due to the increase in the manipulation image size the number of GOBS and the number of macro-blocks per GOB have to be modified also. These modifications are as follows. For a CIF image the number of GOBS increase to 14 and GOBS 13 and 14 should contain 44 Macro-blocks instead of 33. This is because there are 64 extra lines in the Y frame, hence extra 4, super-block widths are needed to cover this gap. Similarly, for QCIF frames there will be four GOBS with the last GOB having only 22 macro-blocks. The NTSC frame will be identical to the CIF frame. This implies that the NTSC frame is not suited to this scheme.

The Transform

The old DCT block transform is obviously now replaced by the new SBDCT. The purpose of this transform is to convert the entire image in to the frequency domain in the method described in the previous chapter. Once this is done the complete image DCT can then be split up into the corresponding GOBS and macro-blocks ready for quantisation. The number of levels traversed down the subband pyramid is left up to the user, however, it should be noted that there is a limit to the quartering process after which the frame cannot be divided equally. These maximums can be seen in Table 5.

Image	Level	WID	HIG
CIF-Y	5	11	11
CIF-U	4	11	11
QCIF-Y	4	11	11
QCIF-U	3	11	11

Table 5 : Image Transform Depth Levels, Widths and Heights

Motion Compensation and Estimation

There are a couple of differences that must be implemented in this area. Firstly, Motion Compensated blocks with zero motion vectors and zero difference cannot be allowed, as the SBDCT cannot have missing blocks. Therefore, all such macro-blocks should now use the same difference scheme used by the other motion compensated blocks. This means that the block pointed to by the vector should be subtracted from the current image and placed in the appropriate spot of the work frame.

The second modification is that the motion vector information for the zero area should be set to inter macro-blocks with zero difference.

The VLC Coding scheme

The variable length coding module also has a few changes that need to be exercised. 1. For CIF and NTSC frames, GOBS 13 and 14 should not have any other macro-block addresses other than one, which indicates only a progressive macro-block count from one to 44. This is because numbers between 33 and 44 cannot be represented with the existing macro-block table. 2. For QCIF frames GOB 4 should not accept any more than 22 progressive macro-block addresses.

The Quantising scheme

The quantisation scheme used in the recommendation H.261 can be adequately used in this scheme as it allows for different quantisation values (via MQUANT) to be used on different macro-blocks. However, the way in which this variable quantisation

scheme is used, now becomes more important. It is known that when a block is DCT transformed the low frequencies tend to gather in the top left hand corner of the transformed block. While the high frequencies tend to gather in the bottom left hand corner of the transformed block. It is also known that the high frequency components can be quantised more than the low frequency components without much loss to the image (unless there are high speed movements).

Since the SBDCT process converts the entire frame into the frequency domain, and since it is then sub-divided into many macro-blocks, using the idea from above, harsher (MQUANT = 8 - 31) quantisation can be given to macro blocks that fall in the high frequency bands of the transformed image, while low frequencies can be quantised with lower quantisation values (MQUANT = 1 - 8). If the quantisation scheme is appropriately applied then certain bands can evade transmission (i.e. CBP = 0) and yet supply a good image to the receiver. This is the heart of this digital motion compression coding scheme.

Conclusion

Although this scheme hasn't been implemented still (Explained later), it is a feasible approach in combining SBDCT image coding scheme, with the Recommendation H.261 to provide not only low bit-rates but also better quality video sequences at these low bit-rates.

Chapter VI : Project Review

Background

This project came into being in February 1995. Our main aims for this project was to implement an existing digital video compression scheme that supported bit-rates of up to 64kbps or below, with the goal of improving it in terms of either picture quality or transmission rate. Searches were made through literature and the Internet on and for different compression schemes. As a result, ITU-TS's Recommendation H.261, with it meeting our first criteria and being available at the time, seemed to be the best to begin implementing.

The implementation was to be software and be done on an IBM compatible PC, with either Boland Pascal or C. Since the course chose Pascal as its primary structured teaching language for students in 1992, Boland Pascal was used. This implementation became a huge task on its own, although the initial simple tasks, such as implementing a DCT module etc. were completed easily. It required quite a huge amount of pointer manipulations just to perform the task of loading a frame. Confusion about the structure of the images and how they should be sub-divided to be successfully used made the implementation difficult. This combined with the variation of the bit-stream structure in different implementations, lead to the creation of 10 or so extra versions.

When motion Estimation and compensation was then introduced this caused too many general protection faults. The in built debugger in Turbo Pascal was disabled as the only way more memory for frames could be attained was by switching the program into protected mode (i.e. disable the in built debugger)

The implementation of this program was completed in October 1995. Yes, the encoder, "free of bugs" encoded frames using motion compensation and estimation while the decoder was able to decode the frames encoded by this program. (The Encoder and Decoder Source code can be seen in Appendix A and Appendix B respectively.) The programs follow the structure and operation of the H.261.

H.261 Implementation Specifics (Program Documentation)

As can be seen from Appendix A and B the both the encode and decode programs have many similar structures, however most information about the frame and the sequence is held in the Info_type structure. The other structures are used to define one and two dimensional arrays for the manipulation of images.

The basic flow of the encode program is a follows.

A. Initilise Program - At this stage the program initialises its VLC tables file information (Paths, Sizes) etc.

B. Open the Output file - This is done here as the output bit-stream is kept open till all the frames are processed.

C. For each frame ... ...

D. Allocate the work and current frames. - Two frames are needed, one to carry out the calculations and one to hold the current image. The work frame is made up of integer values as it can have positive or negative values, while the current frame is made up of byte values.

E. Do motion Compensation - If a previous frame does not exist then the current frame is copied into the work frame with all the macro-blocks set to intra. Else the proper motion compensation is carried out and placed in the work frame.

F. DCT Quantise and store - This part performs the DCT quantisation and storing of all the macro-blocks.

G. Inverse Quantise and DCT - this reverses the above process so that a quantised frame can be put as the previous frame.

H. Inverse motion Compensation - this carries out the motion compensation as defined in each macro-block. Needed to reconstruct the previous frame for the next image.

I. De Allocate work frame

J. Allocate Previous Frame - Allocate current frame as previous frame and de-allocate the previous frame if it exist.

K. ..... end for

L. Close the output file.

The basic flow of the decode program is a follows.

A. Initialise the program - At this stage the program initialises its VLC tables file information (Paths, Sizes) etc.

B. Open input file - this remains open till all frames in it are decoded.

C. While not end of file

D. Read the image header

E. Allocate frame size based on this header

F. Loads a frame from input - This goes through the process of decoding the Variable length codes to gather data for the image.

G. Inverse Quantise and IDCT - Each macro-block goes through this.

H. Inverse Motion Compensate - fixes output frame back together.

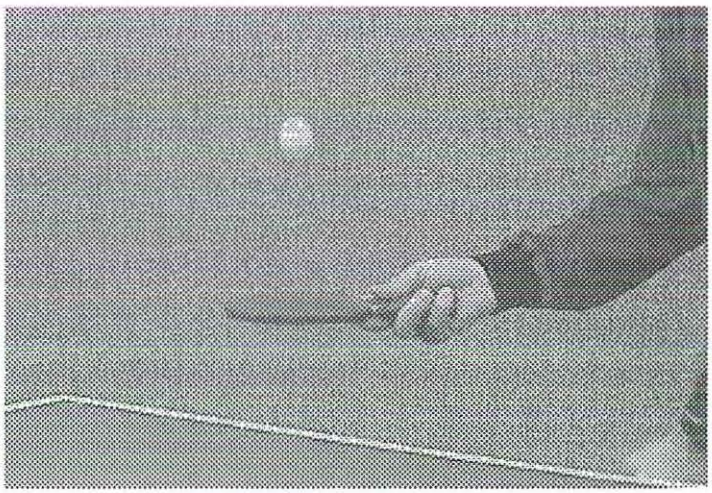
I. Save Frame

J. Copy frame to previous frame.

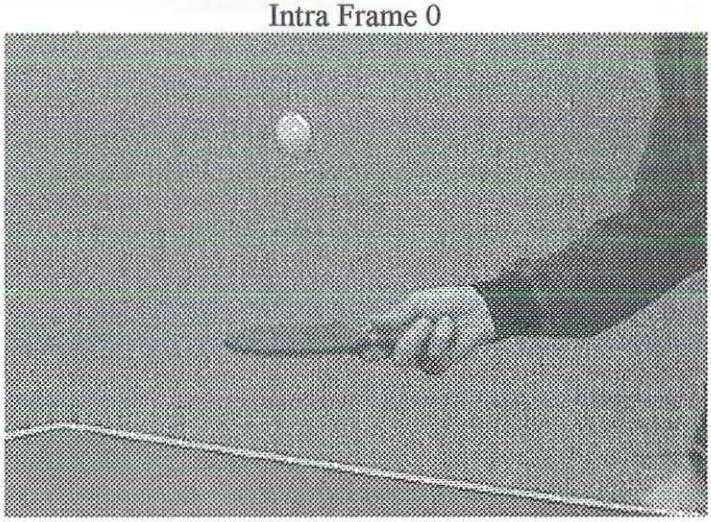
K. ..... end while

L. Close input bit-stream

The following are two frames, figure 15, are encoded and decoded by this program.



Intra Frame 0



Inter Frame 1

Figure 15 : H.261 Implementation Test.

Challenged by the problem of trying to figure out a way to improve this scheme, all subband coding schemes based on QMF filters were closely looked at. However, these were somewhat difficult to directly implement on the H.261 coding scheme. Therefore, the SBDCT scheme was investigated and then chosen as the best alternative. This scheme, together with some modification to the H.261 scheme fulfilled our second goal of improving the implemented standard. However, time ran out on us before it could be implemented.

Chapter VII : Further Improvements and Conclusions

Further Improvements

The new digital motion video coding scheme described in this thesis can be improved in a number of ways. Some of there are discussed below...

1. One of the main problems associated with this scheme is the space wasted by the image enlargement process needed so that the transform can operate. To minimise this one could try modifying the transform structure so that it could be made to handle rectangular images instead of images in a perfect square.
2. Another improvement would be to implement a 3D SBDCT on a block of frames so that costly motion searches can be avoided and more temporal redundancies can be removed.
3. A third way to improve this technique is to replace the SBDCT sub-division in to macro-blocks with a scheme that directly stored the subbands. i.e. not the re-split bands but the initial DST and DCT subbands. However, this approach will also require new Variable Length Coding schemes and tables to be defined.

Conclusion

On the whole this digital video encoding scheme leaves many avenues open for improvement. However it does provide a feasible digital motion video compression implementation scheme that offers bit-rates of 64Kbps or below together with good quality image reconstruction. It therefore can be the basis for a new but related project.

References

[1] Peng H. Ang, Peter A. Ruetz & David Auld (1991) - Video Compression Makes Big Gains - IEEE Spectrum, pp 16-19.

[2] P.M. Bentley & J.T.E. McDonnell (1994) Wavelet Transforms: an Introduction - Electronics & Communication Engineering Journal, pp.175-186

[3] H. Caglar, Y. Liu, & A.N. Akansu (1993) - Optimal PR-QMF Design for Subband Image Coding - Journal of Visual Communication and Image Representation (Volume 4), pp242-253.

[4] Yun-Hee Chan & Wan-Chi Siu (1994) Short Communication: An Approach to Subband DCT Image Coding - Journal of Visual Communication and Image Representation. Academic Press Inc. pp.95-106

[5] Yun-Hee Chan & Wan-Chi Siu - Novel Subband DCT Image Coding, pp61-66.

[6] Cheng-Tie Chen (1993) - Video Compression Standards and Applications - Journal of Visual Communication and Image Representation (Volume 4), pp.103-111.

[7] Tracy Denk, Keshab K. Parhi & Vladimire Cherkassky (1993) Combining Neural Networks and the Wavelet Transform for Image Compression, pp.637-640.

[8] S.C. Huang & Y.F. Huang (1993) - Principal Component Vector Quantization - Journal of Visual Communication and Image Representation, pp. 112-119.

[9] Andy C. Hung (1993) - PVRG-P64 CODEC 1.1.

[10] Emmanuel C. Ifeachor & Barrie W. Jervis (1994) - Digital Signal Processing - A Practical Approach - U.S.: Addison-Wesley Publishers Ltd.

[11] ITU - TS (1990) Video Codec For Audiovisual Services At p x 64 kbit/s (Recommendation H.261).

[12] Anil K. Jain (1981) Image Data Compression: A Review: California: IEEE Computer Society Press, pp411-451.

[13] Xuan Kong & John Goutsias (1994) - Short Communication - A Study of Pyramidal Techniques for Image Representation and Compression - Journal of Visual Communication and Image Representation (Volume 5), pp190-203.

[14] M. L. Liou & H. Fujiwara - VLSI Implementation of a Low Bit-rate Video Codec, pp180-183.

[15] Kwok-Tung Lo & Wai-Kuen Chan (1993) - Short Communication - A Modified Cosine Transform - Journal of Visual Communication and Image Representation (Volume 4), pp 178-185.

[16] King N. Ngan & Weng L. Chooi (1994) Very Low Bit Rate Video Coding Using 3D Subband Approach - IEEE Transaction on Circuits and System for Video Technology (Vol. 4), pp.309-316

[17] Peter Pirsch, Nicolas Demassieux & Winfried Gehrke (1995) VLSI Architectures for Video Compression - A Survey - Proceedings of IEEE Volume 83, pp220-243.

[18] Jens-Rainer Ohm (1994) - three-dimensional Subband Coding with Motion Compensation, pp559-568.

[19] Oliver Rioul and martin Vetterli (1991) - Wavelets and Signal Processing, IEEE SP Magazine October pp 15-38.

[20] Nassrin Tavakoli (1993) - Short Communication - Entropy and Image Compression - Journal of Visual Communication and Image Representation, (Volume 4), pp271-269.

[21] Robert E. Van Dyck & Sarah A. Rajala (1994) - Subband VQ Coding of Color Images Using a Separable Diamond Decomposition - Journal of Visual Communication and Image Representation (Volume 5), pp205-219.

[22] Lora G. Weiss (1994) - Wavelets and Wideband Correlation Processing - IEEE Signal Processing Magazine, pp 13-33.

[23] John W. Woods & Sean D.O'Neil (1986) Subband Coding of Images Califomia: IEEE Computer Society Press, pp.550-560

Appendix A

```
Program H261_Encode(Input,Output);
{*************************************************************************}
{* Program Name : H261_Encode                                            *}
{* Author : Geoffrey Alagoda, Student No : 0911697                       *}
{* Date of Last Modification : 14/October/1995                           *}
{* Purpose : To implement an encoder based on the CCITT H.261 Motion Video *}
{*       Compression / Decompresson Recomendation.  This is done for a   *}
{*       fourth year Engineering Project so that it can be later         *}
{*       improved upon.  The implimentation is to be on an IBM PC        *}
{*       Compatible machine using Tutbo Pascal version 7 with at least   *}
{*       8 Mb of RAM and > 386 DX40.  The final Encoded file produced    *}
{*       may differ slightly from other implementation as its based on   *}
{*       my interpretation of the H.261 Standard.                        *}
{*                                                                       *}
{* Program Modules: Motion Compensation / Estimation                     *}
{*             DCT / IDCT                                                *}
{*             Quantisation / De-Quantisation                            *}
{*             Run Length Encoding                                       *}
{*             Variable Length Encoding                                  *}
{*             File Management Code                                      *}
{*             Memory Management Code - etc                              *}
{*************************************************************************}

uses crt,dos;          {* Standard Turbo Pascal Screen and File Library *}

const
   BLK_WID = 8;            {* The width of a block to be dealt with. *}
   BLK_HIG = 8;           {* The height of a block to be dealt with. *}

   GOB_WID = 11;                    {* MAC Blocks Per GOB Row *}
   GOB_HIG = 3;                    {* MAC Blocks Per GOB Col *}
   MAC_GOB = GOB_WID * GOB_HIG;       {* Number of Macroblocks per GOB *}

   MAX_GOBS = 12;                    {* Maximum Number Of GOBS *}
   MAX_CBP = 6;                    {* Max Blocks per Macroblock *}
   SRCH_WID = 15;                    {* MC Pel Search Size *}
   ZRO_POS = 3 * BLK_WID * 4 * BLK_HIG + BLK_WID * 2 + 1;
   ZRO_POS1 = 3 * BLK_WID * BLK_HIG + BLK_WID + 1;

   {* Picture Size Definitions *}

   P_CIF_YWID = 352;
   P_CIF_YHIG = 288;
   P_CIF_CWID = 176;
   P_CIF_CHIG = 144;
   CIF_GOB = 12;

   P_QCIF_YWID = 176;
   P_QCIF_YHIG = 144;
   P_QCIF_CWID = 88;
   P_QCIF_CHIG = 72;
   QCIF_GOB = 3;

   N_CIF_YWID = 352;
   N_CIF_YHIG = 240;
   N_CIF_CWID = 176;
   N_CIF_CHIG = 120;
   NTSC_GOB = 10;
```

Appendix A

```
{* Picture Size Definitions *}

{* Quantisation *}

IGQUANT = 8;
IMQUANT = 8;
DCQUANT = IGQUANT;

{* Picture Types *}

CIF = 1;                                {* 12 GOBS *}
QCIF = 2;                               {* 3 GOBS *}
NTSC = 3;                               {* 10 GOBS *}

{* Picture Types *}

type
  Rawpicvaltype = byte;        {* For Raw Picture Loading - saves memory *}
  Picvaltype = integer;        {* The Value Type For A Picture Level *}
  DCTvaltype = single;         {* The Value Type For A DCT Level *}


  Picblktype = array[1..BLK_WID,1..BLK_HIG] of Picvaltype;
  Rpicblktype = array[1..BLK_WID,1..BLK_HIG] of Rawpicvaltype;
                                {* Block Type Dec. *}
  Spicblktype = array[1..BLK_WID*2,1..BLK_HIG*2] of Picvaltype;
  Rspicblktype = array[1..BLK_WID*2,1..BLK_HIG*2] of Rawpicvaltype;
                                {* Super Block Type Dec. *}
  Blklinetype = array[1..BLK_WID*BLK_HIG] of Picvaltype;
                                {* Line After ZigZag Type Dec. *}
  Ftype = file of byte;                 {* Stream File Type *}

  Rmacptrtype = ^Rmactype;          {* Used to define Blocks with Bytes *}
  Rmactype = record
       RawMacYArray : Rspicblktype;
       RawMacUArray : Rpicblktype;
       RawMacVArray : Rpicblktype;
  end;
  Macblkptrtype = ^Macblktype;
                {* Used For Differential Blocks with integers *}
  Macblktype = record
        MType : byte;        {* Macro Blk type as in Table 2/H.261 *}
        MQuant : byte;      {* Quantisation if Mtype in 2,4,7 or 10 *}
        MVV,MVH : shortint;                 {* Motion vector *}
        CBP : shortint;                {* Which To transmit *}
                              {* Blocks For Pic Info *}
        RawMacYArray : Spicblktype;
        RawMacUArray : Picblktype;
        RawMacVArray : Picblktype;
  end;

  Gobptrtype = ^Gobtype;
  Gobtype = record                      {* Group Of Blocks Type *}
      GQuant : byte;                    {* GOB Quantisation *}
      Macs : Array [1..MAC_GOB] of Macblktype;      {* Macro Blocks *}

  end;
```

Appendix A

```
Rgobptrtype = ^RGobtype;
RGobtype = record                          {* Group Of Blocks Type *}
      Macs : Array [1..MAC_GOB] of RMactype;        {* Macro Blocks *}

end;


Pictype = array[1..MAX_GOBS] of Gobptrtype;    {* An Integer work Frame *}
Rpictype = array[1..MAX_GOBS] of Rgobptrtype;      {* A raw byte frame *}
Mcatype = array [1..9] of Rmacptrtype;  {* For MC The Surrounding Blocks *}

Bstype = record                                {* Bit Stream File *}
     VBits : byte;                    {* The Number Of Valid Bits *}
     CBits : byte;
     ChkVal : string;                      {* Not used In Encoding *}
     Tr_Bits : string;                  {* The Actual Transfere Bits *}
                    {* Not Used In Encoding Only For Decoding *}
     FEOF : boolean;
     BsFile : Ftype;
     FSize : single;
     CPos : single;
end;

Vlccoftype = record        {* Variable length Encoding For Coefficients *}
     COFF : string;
     Run : byte;
     Level : byte;
end;

Zztype = Array[1..63] of byte;
                  {* Zig Zag Lookup Table - Easier and Quicker *}

Infotype = record      {* Generally Contains Most Info About Everything *}
     L_File_Name,C_File_Name : string;
                    {* The Loose and Compressed File names *}
     Start_Fr,End_Fr : word;   {* Start Frame and End Frame Numbers *}
     Mov_Type : byte;            {* 2 = QCIF, 1 = CIF & 3 = NTSC *}
     P_Mov_Type : byte;              {* Previous Frame Mov_Type *}

     Split_Screen : Boolean;           {* Split Screen Indicator *}
     Doc_Cam_Ind : Boolean;                {* Camera Indicator *}
     Fr_Pic_Rel : Boolean;             {* Reletive Pic Indicator *}
     TRef : Byte;                      {* Temporal Reference*}

     MBATab : Array[1..34] of string;          {* MBA Lookup Table *}
     MTypeTab : Array[1..10] of string;       {* MType Lookup Table *}
     MVDTab : Array[0..31] of string;    {* Motion Vec Lookup Table *}
     CBPTab : Array[1..63] of string;          {* Transfer Sequence *}
     COTab : Array[1..63] of Vlccoftype;      {* Coef. Lookup table *}
     ZZCol : Zztype;                  {* ZiZag Col. Lookup Table *}
     ZZRow : Zztype;                   {* ZiZag Row. Lookup Table *}
     Debug : boolean;
     DebugVal : byte;
end;                                        {* Infotype Defn *}


var                                         {* Global Variables *}
```

```
  Info : Infotype;                              {* Main Info *}

{* ------------------------------------------------------------ *}
function FFName (InStr : string; Value : word; Ext : Char) : string;
{* This function Constructs filename strings for Loose files with values at *}
{* the end of the name                                          *}

var
  TmpStr : String;

begin
  str(Value:0,TmpStr);
  FFName := InStr+TmpStr+'.'+Ext;
end;

function FileExists (Nam : string) : Single;
{* This function looks to see if a file exists and returns the file size.  *}

var
  F : file of byte;
  At : word;
  F_Size : longint;

begin
  assign(F,Nam);
  GetFAttr(F, At);
  if DOSERROR <> 0 then
    FileExists := 0
  else
    begin
      reset(F);
      F_Size := filesize(F);
      FileExists := F_Size;
    end;
  if DOSERROR = 0 then
    close(F);
end;

function FrameID(FS : Single; Ch : Char): byte;
{* This Function looks at the size of the file and determines if the frame *}
{* is 1 = PAL CIF, 2 = PAL QCIF, 3 = NTSC CIF                              *}

begin
  case Ch of
    'Y' : if FS = (P_CIF_YWID * P_CIF_YHIG) then
            FrameID := CIF
          else
            if FS = (P_QCIF_YWID * P_QCIF_YHIG) then
              FrameID := QCIF
            else
              if FS = (N_CIF_YWID * N_CIF_YHIG) then
                FrameID := NTSC
              else
                FrameID := 0;

    'U' : if FS = (P_CIF_CWID * P_CIF_CHIG) then
            FrameID := CIF
          else
```

```
          if FS = (P_QCIF_CWID * P_QCIF_CHIG) then
            FrameID := QCIF
          else
            if FS = (N_CIF_CWID * N_CIF_CHIG) then
              FrameID := NTSC
            else
              FrameID := 0;

    'V' : if FS = (P_CIF_CWID * P_CIF_CHIG) then
          FrameID := CIF
        else
          if FS = (P_QCIF_CWID * P_QCIF_CHIG) then
            FrameID := QCIF
          else
            if FS = (N_CIF_CWID * N_CIF_CHIG) then
              FrameID := NTSC
            else
              FrameID := 0;
  end;

end;

procedure Die(Msg : string ; Code : byte);
{* This procedure is to give an error message when an error occurs and     *}
{* un-gracefully Quit The program with the error code.                     *}

begin
  writeln;
  writeln;
  if Msg = '' then
    writeln('Error - Dowh, this is new one, dunno what this is')
  else
    writeln('Error ',Code,': ',Msg);
  write('Program Crashed, Please Hit Enter ->');
  readln;
  halt(Code);
end;

function BtS(InVal : Byte) : string;
{* This Function converts InVal in to a string of bits.                   *}

var
  str1 : string[8];


begin
  str1 := '';
  if (InVal and 128) = 128 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 64) = 64 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 32) = 32 then
    str1 := str1 + '1'
  else
```

```
    str1 := str1 + '0';
  if (InVal and 16) = 16 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 8) = 8 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 4) = 4 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 2) = 2 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 1) = 1 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';

  BtS := str1;
end;


function StB(InVal : String) : byte;
{* This Function Converts the InVal String of bits to a byte.          *}

var
  Out : byte;
  Idx1 : byte;
  Len : Byte;
  TmpStr : string;


begin
  Out := 0;
  Len := Length(InVal);
  TmpStr := '';

  for Idx1 := 1 to (8 - Len) do
    TmpStr := TmpStr + '0';
  TmpStr := TmpStr + InVal;

  for Idx1 := 1 to 8 do
  if TmpStr[Idx1] = '1' then
    case  Idx1 of
      1 : Out := Out + 128;
      2 : Out := Out + 64;
      3 : Out := Out + 32;
      4 : Out := Out + 16;
      5 : Out := Out + 8;
      6 : Out := Out + 4;
      7 : Out := Out + 2;
      8 : Out := Out + 1;
    end;
  StB := Out;
```

```
end;

{* ---------------------------------------------------------- *}


procedure Init(var I : Infotype);
{* This Procedure Initialises the Main Info with Data such as VLC Table    *}
{* Codes, Filenames.. etc.                                                 *}

var
  InStr : string;
  Result : integer;
  Idx : word;
  FExists : boolean;
  FSY,FSU,FSV : Single;                    {* File Size Y,U,V *}
  FID : byte;

begin
  clrscr;                {* Turbo Pascal's Famous Clear Screen Routine *}
  writeln('H.261 Motion Video Compression Encoder V 0.99 (Geoffrey Alagoda - 1995)');
  writeln;
  repeat
    writeln('Please Enter Uncompressed File Name common to all Frames');
    write('->');
    {readln(InStr);}
    InStr := 'C:\APPS\WORK\PROJ\C\SHORT';
    writeln(InStr);
  until InStr <> '';
  I.L_File_Name := InStr;
  repeat
    write('Please Enter The Starting Frame Number ->');
    InStr := '0';
    writeln(InStr);
    {readln(InStr);}
    val(InStr,I.Start_Fr,Result);
  until (InStr <> '') and (Result = 0);
  repeat
    write('Please Enter Ending Frame Number >= Starting Frame ->');
    InStr := '6';
    writeln(InStr);
    {readln(InStr);}
    val(InStr,I.End_Fr,Result);
  until (InStr <> '') and (Result = 0) and (I.End_Fr >= I.Start_Fr);

  FExists := true;
  Idx := I.Start_Fr;

  while (FExists) and (Idx <= I.End_Fr) do
    begin
      FSY := FileExists(FFName(I.L_File_Name,Idx,'Y'));
      FSU := FileExists(FFName(I.L_File_Name,Idx,'U'));
      FSV := FileExists(FFName(I.L_File_Name,Idx,'V'));
      FID := FrameID(FSY,'Y');
      if (FID = FrameID(FSU,'U')) and (FID = FrameID(FSV,'V')) then
        if FID in [CIF,QCIF,NTSC] then
          writeln('Frame ',Idx,' : ',FID)
        else
          FExists := False
```

```
    else
      FExists := False;
    Idx := Idx + 1;
  end;

If Not FExists then
  Die('All Loose Files NOT Found or NOT of the Same Type',1);

I.Mov_Type := FID;
repeat
  write('Please Enter Output File Name ->');
  InStr := 'C:\APPS\WORK\PROJ\C\Test1.261';
  {readln(InStr);}
  writeln(InStr);
until (InStr <> '');
I.C_File_Name := InStr;

I.Debug := False;
I.DebugVal := 0;

writeln('Initialising Tables');

I.MBATab[1] := '1';
I.MBATab[2] := '011';
I.MBATab[3] := '010';
I.MBATab[4] := '0011';
I.MBATab[5] := '0010';
I.MBATab[6] := '00011';
I.MBATab[7] := '00010';
I.MBATab[8] := '0000111';
I.MBATab[9] := '0000110';
I.MBATab[10] := '00001011';
I.MBATab[11] := '00001010';
I.MBATab[12] := '00001001';
I.MBATab[13] := '00001000';
I.MBATab[14] := '00000111';
I.MBATab[15] := '00000110';
I.MBATab[16] := '0000010111';
I.MBATab[17] := '0000010110';
I.MBATab[18] := '0000010101';
I.MBATab[19] := '0000010100';
I.MBATab[20] := '0000010011';
I.MBATab[21] := '0000010010';
I.MBATab[22] := '00000100011';
I.MBATab[23] := '00000100010';
I.MBATab[24] := '00000100001';
I.MBATab[25] := '00000100000';
I.MBATab[26] := '00000011111';
I.MBATab[27] := '00000011110';
I.MBATab[28] := '00000011101';
I.MBATab[29] := '00000011100';
I.MBATab[30] := '00000011011';
I.MBATab[31] := '00000011010';
I.MBATab[32] := '00000011001';
I.MBATab[33] := '00000011000';
I.MBATab[34] := '00000001111';

I.MTypeTab[1] := '0001';
```

Appendix A

```
I.MTypeTab[2] := '0000001';
I.MTypeTab[3] := '1';
I.MTypeTab[4] := '00001';
I.MTypeTab[5] := '000000001';
I.MTypeTab[6] := '00000001';
I.MTypeTab[7] := '0000000001';
I.MTypeTab[8] := '001';
I.MTypeTab[9] := '01';
I.MTypeTab[10] := '000001';

I.MVDTab[0] := '1';
I.MVDTab[1] := '010';
I.MVDTab[2] := '0010';
I.MVDTab[3] := '00010';
I.MVDTab[4] := '0000110';
I.MVDTab[5] := '00001010';
I.MVDTab[6] := '00001000';
I.MVDTab[7] := '00000110';
I.MVDTab[8] := '0000010110';
I.MVDTab[9] := '0000010100';
I.MVDTab[10] := '0000010010';
I.MVDTab[11] := '00000100010';
I.MVDTab[12] := '00000100000';
I.MVDTab[13] := '00000011110';
I.MVDTab[14] := '00000011100';
I.MVDTab[15] := '00000011010';
I.MVDTab[16] := '00000011001';
I.MVDTab[17] := '00000011011';
I.MVDTab[18] := '00000011101';
I.MVDTab[19] := '00000011111';
I.MVDTab[20] := '00000100001';
I.MVDTab[21] := '00000100011';
I.MVDTab[22] := '0000010011';
I.MVDTab[23] := '0000010101';
I.MVDTab[24] := '0000010111';
I.MVDTab[25] := '00000111';
I.MVDTab[26] := '00001001';
I.MVDTab[27] := '00001011';
I.MVDTab[28] := '0000111';
I.MVDTab[29] := '00011';
I.MVDTab[30] := '0011';
I.MVDTab[31] := '011';

I.CBPTab[60] := '111';
I.CBPTab[4] := '1101';
I.CBPTab[8] := '1100';
I.CBPTab[16] := '1011';
I.CBPTab[32] := '1010';
I.CBPTab[12] := '10011';
I.CBPTab[48] := '10010';
I.CBPTab[20] := '10001';
I.CBPTab[40] := '10000';
I.CBPTab[28] := '01111';
I.CBPTab[44] := '01110';
I.CBPTab[52] := '01101';
I.CBPTab[56] := '01100';
I.CBPTab[1] := '01011';
I.CBPTab[61] := '01010';
```

Appendix A

```
I.CBPTab[2] := '01001' ;
I.CBPTab[62] := '01000';
I.CBPTab[24] := '001111';
I.CBPTab[36] := '001110';
I.CBPTab[3] := '001101' ;
I.CBPTab[63] := '001100';
I.CBPTab[5] := '0010111';
I.CBPTab[9] := '0010110';
I.CBPTab[17] := '0010101';
I.CBPTab[33] := '0010100';
I.CBPTab[6] := '0010011' ;
I.CBPTab[10] := '0010010';
I.CBPTab[18] := '0010001';
I.CBPTab[34] := '0010000';
I.CBPTab[7] := '00011111';
I.CBPTab[11] := '00011110';
I.CBPTab[19] := '00011101';
I.CBPTab[35] := '00011100';
I.CBPTab[13] := '00011011';
I.CBPTab[49] := '00011010';
I.CBPTab[21] := '00011001';
I.CBPTab[41] := '00011000';
I.CBPTab[14] := '00010111';
I.CBPTab[50] := '00010110';
I.CBPTab[22] := '00010101';
I.CBPTab[42] := '00010100';
I.CBPTab[15] := '00010011';
I.CBPTab[51] := '00010010';
I.CBPTab[23] := '00010001';
I.CBPTab[43] := '00010000';
I.CBPTab[25] := '00001111';
I.CBPTab[37] := '00001110';
I.CBPTab[26] := '00001101';
I.CBPTab[38] := '00001100';
I.CBPTab[29] := '00001011';
I.CBPTab[45] := '00001010';
I.CBPTab[53] := '00001001';
I.CBPTab[57] := '00001000';
I.CBPTab[30] := '00000111';
I.CBPTab[46] := '00000110';
I.CBPTab[54] := '00000101';
I.CBPTab[58] := '00000100';
I.CBPTab[31] := '000000111';
I.CBPTab[47] := '000000110';
I.CBPTab[55] := '000000101';
I.CBPTab[59] := '000000100';
I.CBPTab[27] := '000000011';
I.CBPTab[39] := '000000010';


(I.COTab[1]).COFF := '000001';
(I.COTab[1]).Run := 27;
(I.COTab[1]).Level := 1;

(I.COTab[2]).COFF := '011';
(I.COTab[2]).Run := 1;
(I.COTab[2]).Level := 1;
```

Appendix A

```
(I.COTab[3]).COFF := '0100';
(I.COTab[3]).Run := 0;
(I.COTab[3]).Level := 2;

(I.COTab[4]).COFF := '0101';
(I.COTab[4]).Run := 2;
(I.COTab[4]).Level := 1;

(I.COTab[5]).COFF := '00101';
(I.COTab[5]).Run := 0;
(I.COTab[5]).Level := 3;

(I.COTab[6]).COFF := '00111';
(I.COTab[6]).Run := 3;
(I.COTab[6]).Level := 1;

(I.COTab[7]).COFF := '00110';
(I.COTab[7]).Run := 4;
(I.COTab[7]).Level := 1;

(I.COTab[8]).COFF := '000110';
(I.COTab[8]).Run := 1;
(I.COTab[8]).Level := 2;

(I.COTab[9]).COFF := '000111';
(I.COTab[9]).Run := 5;
(I.COTab[9]).Level := 1;

(I.COTab[10]).COFF := '000101';
(I.COTab[10]).Run := 6;
(I.COTab[10]).Level := 1;

(I.COTab[11]).COFF := '000100';
(I.COTab[11]).Run := 7;
(I.COTab[11]).Level := 1;

(I.COTab[12]).COFF := '0000110';
(I.COTab[12]).Run := 0;
(I.COTab[12]).Level := 4;

(I.COTab[13]).COFF := '0000100';
(I.COTab[13]).Run := 2;
(I.COTab[13]).Level := 2;

(I.COTab[14]).COFF := '0000111';
(I.COTab[14]).Run := 8;
(I.COTab[14]).Level := 1;

(I.COTab[15]).COFF := '0000101';
(I.COTab[15]).Run := 9;
(I.COTab[15]).Level := 1;

(I.COTab[16]).COFF := '00100110';
(I.COTab[16]).Run := 0;
(I.COTab[16]).Level := 5;

(I.COTab[17]).COFF := '00100001';
(I.COTab[17]).Run := 0;
```

```
(I.COTab[17]).Level := 6;

(I.COTab[18]).COFF := '0010010i';
(I.COTab[18]).Run := 1;
(I.COTab[18]).Level := 3;

(I.COTab[19]).COFF := '00100100';
(I.COTab[19]).Run := 3;
(I.COTab[19]).Level := 2;

(I.COTab[20]).COFF := '00100111';
(I.COTab[20]).Run := 10;
(I.COTab[20]).Level := 1;

(I.COTab[21]).COFF := '00100011';
(I.COTab[21]).Run := 11;
(I.COTab[21]).Level := 1;

(I.COTab[22]).COFF := '00100010';
(I.COTab[22]).Run := 12;
(I.COTab[22]).Level := 1;

(I.COTab[23]).COFF := '00100000';
(I.COTab[23]).Run := 13;
(I.COTab[23]).Level := 1;

(I.COTab[24]).COFF := '0000001010';
(I.COTab[24]).Run := 0;
(I.COTab[24]).Level := 7;

(I.COTab[25]).COFF := '0000001100';
(I.COTab[25]).Run := 1;
(I.COTab[25]).Level := 4;

(I.COTab[26]).COFF := '0000001011';
(I.COTab[26]).Run := 2;
(I.COTab[26]).Level := 3;

(I.COTab[27]).COFF := '0000001111';
(I.COTab[27]).Run := 4;
(I.COTab[27]).Level := 2;

(I.COTab[28]).COFF := '0000001001';
(I.COTab[28]).Run := 5;
(I.COTab[28]).Level := 2;

(I.COTab[29]).COFF := '0000001110';
(I.COTab[29]).Run := 14;
(I.COTab[29]).Level := 1;

(I.COTab[30]).COFF := '0000001101';
(I.COTab[30]).Run := 15;
(I.COTab[30]).Level := 1;

(I.COTab[31]).COFF := '0000001000';
(I.COTab[31]).Run := 16;
(I.COTab[31]).Level := 1;
```

```
(I.COTab[32]).COFF := '000000011101';
(I.COTab[32]).Run := 0;
(I.COTab[32]).Level := 8;

(I.COTab[33]).COFF := '000000011000';
(I.COTab[33]).Run := 0;
(I.COTab[33]).Level := 9;

(I.COTab[34]).COFF := '000000010011';
(I.COTab[34]).Run := 0;
(I.COTab[34]).Level := 10;

(I.COTab[35]).COFF := '000000010000';
(I.COTab[35]).Run := 0;
(I.COTab[35]).Level := 11;

(I.COTab[36]).COFF := '000000011011';
(I.COTab[36]).Run := 1;
(I.COTab[36]).Level := 5;

(I.COTab[37]).COFF := '000000010100';
(I.COTab[37]).Run := 2;
(I.COTab[37]).Level := 4;

(I.COTab[38]).COFF := '000000011100';
(I.COTab[38]).Run := 3;
(I.COTab[38]).Level := 3;

(I.COTab[39]).COFF := '000000010010';
(I.COTab[39]).Run := 4;
(I.COTab[39]).Level := 3;

(I.COTab[40]).COFF := '000000011110';
(I.COTab[40]).Run := 6;
(I.COTab[40]).Level := 2;

(I.COTab[41]).COFF := '000000010101';
(I.COTab[41]).Run := 7;
(I.COTab[41]).Level := 2;

(I.COTab[42]).COFF := '000000010001';
(I.COTab[42]).Run := 8;
(I.COTab[42]).Level := 2;

(I.COTab[43]).COFF := '000000011111';
(I.COTab[43]).Run := 17;
(I.COTab[43]).Level := 1;

(I.COTab[44]).COFF := '000000011010';
(I.COTab[44]).Run := 18;
(I.COTab[44]).Level := 1;

(I.COTab[45]).COFF := '000000011001';
(I.COTab[45]).Run := 19;
(I.COTab[45]).Level := 1;

(I.COTab[46]).COFF := '000000010111';
(I.COTab[46]).Run := 20;
```

```
(I.COTab[46]).Level := 1;

(I.COTab[47]).COFF :='000000010110';
(I.COTab[47]).Run := 21;
(I.COTab[47]).Level := 1;

(I.COTab[48]).COFF := '0000000011010';
(I.COTab[48]).Run := 0;
(I.COTab[48]).Level := 12;

(I.COTab[49]).COFF := '0000000011001';
(I.COTab[49]).Run := 0;
(I.COTab[49]).Level := 13;

(I.COTab[50]).COFF := '0000000011000';
(I.COTab[50]).Run := 0;
(I.COTab[50]).Level := 14;

(I.COTab[51]).COFF := '0000000010111';
(I.COTab[51]).Run := 0;
(I.COTab[51]).Level := 15;

(I.COTab[52]).COFF := '0000000010110';
(I.COTab[52]).Run := 1;
(I.COTab[52]).Level := 6;

(I.COTab[53]).COFF := '0000000010101';
(I.COTab[53]).Run := 1;
(I.COTab[53]).Level := 7;

(I.COTab[54]).COFF := '0000000010100';
(I.COTab[54]).Run := 2;
(I.COTab[54]).Level := 5;

(I.COTab[55]).COFF := '0000000010011';
(I.COTab[55]).Run := 3;
(I.COTab[55]).Level := 4;

(I.COTab[56]).COFF := '0000000010010';
(I.COTab[56]).Run := 5;
(I.COTab[56]).Level := 3;

(I.COTab[57]).COFF := '0000000010001';
(I.COTab[57]).Run := 9;
(I.COTab[57]).Level := 2;

(I.COTab[58]).COFF := '0000000010000';
(I.COTab[58]).Run := 10;
(I.COTab[58]).Level := 2;

(I.COTab[59]).COFF := '0000000011111';
(I.COTab[59]).Run := 22;
(I.COTab[59]).Level := 1;

(I.COTab[60]).COFF := '0000000011110';
(I.COTab[60]).Run := 23;
(I.COTab[60]).Level := 1;
```

Appendix A

```
(I.COTab[61]).COFF := '0000000011101';
(I.COTab[61]).Run := 24;
(I.COTab[61]).Level := 1;

(I.COTab[62]).COFF := '0000000011100';
(I.COTab[62]).Run := 25;
(I.COTab[62]).Level := 1;

(I.COTab[63]).COFF := '0000000011011';
(I.COTab[63]).Run := 26;
(I.COTab[63]).Level := 1;


I.ZZRow[1] := 0; I.ZZCol[1] := 1;
I.ZZRow[2] := 1; I.ZZCol[2] := 0;
I.ZZRow[3] := 2; I.ZZCol[3] := 0;
I.ZZRow[4] := 1; I.ZZCol[4] := 1;
I.ZZRow[5] := 0; I.ZZCol[5] := 2;
I.ZZRow[6] := 0; I.ZZCol[6] := 3;
I.ZZRow[7] := 1; I.ZZCol[7] := 2;
I.ZZRow[8] := 2; I.ZZCol[8] := 1;
I.ZZRow[9] := 3; I.ZZCol[9] := 0;
I.ZZRow[10] := 4; I.ZZCol[10] := 0;
I.ZZRow[11] := 3; I.ZZCol[11] := 1;
I.ZZRow[12] := 2; I.ZZCol[12] := 2;
I.ZZRow[13] := 1; I.ZZCol[13] := 3;
I.ZZRow[14] := 0; I.ZZCol[14] := 4;
I.ZZRow[15] := 0; I.ZZCol[15] := 5;
I.ZZRow[16] := 1; I.ZZCol[16] := 4;
I.ZZRow[17] := 2; I.ZZCol[17] := 3;
I.ZZRow[18] := 3; I.ZZCol[18] := 2;
I.ZZRow[19] := 4; I.ZZCol[19] := 1;
I.ZZRow[20] := 5; I.ZZCol[20] := 0;
I.ZZRow[21] := 6; I.ZZCol[21] := 0;
I.ZZRow[22] := 5; I.ZZCol[22] := 1;
I.ZZRow[23] := 4; I.ZZCol[23] := 2;
I.ZZRow[24] := 3; I.ZZCol[24] := 3;
I.ZZRow[25] := 2; I.ZZCol[25] := 4;
I.ZZRow[26] := 1; I.ZZCol[26] := 5;
I.ZZRow[27] := 0; I.ZZCol[27] := 6;
I.ZZRow[28] := 0; I.ZZCol[28] := 7;
I.ZZRow[29] := 1; I.ZZCol[29] := 6;
I.ZZRow[30] := 2; I.ZZCol[30] := 5;
I.ZZRow[31] := 3; I.ZZCol[31] := 4;
I.ZZRow[32] := 4; I.ZZCol[32] := 3;
I.ZZRow[33] := 5; I.ZZCol[33] := 2;
I.ZZRow[34] := 6; I.ZZCol[34] := 1;
I.ZZRow[35] := 7; I.ZZCol[35] := 0;
I.ZZRow[36] := 7; I.ZZCol[36] := 1;
I.ZZRow[37] := 6; I.ZZCol[37] := 2;
I.ZZRow[38] := 5; I.ZZCol[38] := 3;
I.ZZRow[39] := 4; I.ZZCol[39] := 4;
I.ZZRow[40] := 3; I.ZZCol[40] := 5;
I.ZZRow[41] := 2; I.ZZCol[41] := 6;
I.ZZRow[42] := 1; I.ZZCol[42] := 7;
I.ZZRow[43] := 2; I.ZZCol[43] := 7;
I.ZZRow[44] := 3; I.ZZCol[44] := 6;
I.ZZRow[45] := 4; I.ZZCol[45] := 5;
```

```
  I.ZZRow[46] := 5; I.ZZCol[46] := 4;
  I.ZZRow[47] := 6; I.ZZCol[47] := 3;
  I.ZZRow[48] := 7; I.ZZCol[48] := 2;
  I.ZZRow[49] := 7; I.ZZCol[49] := 3;
  I.ZZRow[50] := 6; I.ZZCol[50] := 4;
  I.ZZRow[51] := 5; I.ZZCol[51] := 5;
  I.ZZRow[52] := 4; I.ZZCol[52] := 6;
  I.ZZRow[53] := 3; I.ZZCol[53] := 7;
  I.ZZRow[54] := 4; I.ZZCol[54] := 7;
  I.ZZRow[55] := 5; I.ZZCol[55] := 6;
  I.ZZRow[56] := 6; I.ZZCol[56] := 5;
  I.ZZRow[57] := 7; I.ZZCol[57] := 4;
  I.ZZRow[58] := 7; I.ZZCol[58] := 5;
  I.ZZRow[59] := 6; I.ZZCol[59] := 6;
  I.ZZRow[60] := 5; I.ZZCol[60] := 7;
  I.ZZRow[61] := 6; I.ZZCol[61] := 7;
  I.ZZRow[62] := 7; I.ZZCol[62] := 6;
  I.ZZRow[63] := 7; I.ZZCol[63] := 7;

  writeln('Tables Initialised');
end;

function Fr_Chars (MType : word;Ch : Char) : word;
{* This Function Returns Characteristics About A Frame defined by Ch and    *}
{* Mtype ie # of GOBS or # of MACs per frame.                                *}

begin
  Fr_Chars := 0;
  if Ch = 'M' then
    begin
      if MType = CIF then
        Fr_Chars := MAC_GOB * 12;
      if MType = QCIF then
        Fr_Chars := MAC_GOB * 3;
      if MType = NTSC then
        Fr_Chars := MAC_GOB * 10;
    end;

  if Ch = 'G' then
    begin
      if MType = CIF then
        Fr_Chars := 12;
      if MType = QCIF then
        Fr_Chars := 3;
      if MType = NTSC then
        Fr_Chars := 10;
    end;


end;

Procedure AllocFrame(var Pic : PicType; MovType : byte);
{* This Procedure Allocates working integer space for a frame.              *}

var
  Idx1,Idx2 : word;

begin
```

```
  for Idx1 := 1 to Fr_Chars(MovType,'G') do
    begin
      new(Pic[Idx1]);
      Pic[Idx1]^.GQuant := IGQUANT;
    end;
end;

procedure UnloadFrame(var I : Infotype; var Pic : Pictype);
{* This Procedure De-allocates a working Integer frame.          *}

var
  Idx1,Idx2 : word;

begin
  for Idx1 := 1 to Fr_Chars(I.Mov_Type,'G') do
    begin
      dispose(Pic[Idx1]);
    end;
end;

Procedure RAllocFrame(var Pic : RpicType; MovType : byte);
{* This Procedure Allocates A Raw Byte Frame To Load-Save A Loose Picture.  *}

var
  Idx1,Idx2 : word;

begin
  for Idx1 := 1 to Fr_Chars(MovType,'G') do
    begin
      new(Pic[Idx1]);
    end;
end;

procedure RUnloadFrame(var I : Infotype; var Pic : RPictype);
{* This Procedure DEAllocates A Raw Byte Frame To Load-Save A Loose Picture.*}

var
  Idx1,Idx2 : word;

begin
  for Idx1 := 1 to Fr_Chars(I.Mov_Type,'G') do
    begin
      dispose(Pic[Idx1]);
    end;
end;


procedure LoadFrame(Var I : Infotype; var Pic : Rpictype; FNum : word);
{* This Procedure Callse Allocate and loads a Frame into Pic.          *}

var
  Idx1,Idx2,Idx3,Count,Blah1,Blah2 : word;
  MovType : byte;
  NumGOBS,NumMACS : word;
  MAC,MACRaw,MACRow,GOB,TrRow,CLine,CLine2 : word;
  InY : Rspicblktype;
  InU,InV : Rpicblktype;
```

```
  YFile : file of Rspicblktype;
  UFile : file of Rpicblktype;
  VFile : file of Rpicblktype;


begin
  MovType := FrameID(FileExists(FFName(I.L_File_name,FNum,'Y')),'Y');

  if not (MovType in [CIF,QCIF,NTSC]) then
    Die('Cannot Identify Frame (LoadFrame)',2);

  NumGOBS := Fr_Chars(MovType,'G');
  NumMACS := Fr_Chars(MovType,'M');

  assign(YFile,FFName(I.L_File_name,FNum,'Y'));
  assign(UFile,FFName(I.L_File_name,FNum,'U'));
  assign(VFile,FFName(I.L_File_name,FNum,'V'));

  reset(YFile);
  reset(UFile);
  reset(VFile);

  RAllocFrame(Pic,Movtype);

  Count := 1;
  for Idx1 := 1 to NumMACS do
    begin
      read(YFile,InY);                          {* Load Y Frame *}
      for Idx2 := 1 to (BLK_WID*2) do
        begin

{* This Fiddely Bit Serves to copy a 16 x 16 Super block into the raw Frame *}
{* Because When its loaded the bytes are not arranged in blocks but in a    *}
{* Sequence, which gets placed in the load 16x16 block. Therefor this part  *}
{* is required to convert this block in to the large frame size.            *}

          if MovType in [CIF,NTSC] then
            begin
              TrRow := ((Count - 1) div (GOB_WID * 2)) + 1;
              MACRaw := ((Count - 1) mod (GOB_WID * 2)) + 1;

              CLine := ((TrRow - 1) mod (BLK_WID * 2)) + 1;
              CLine2 := ((TrRow - 1) mod (BLK_WID)) + 1;
              MACRow := ((TrRow - 1) div (BLK_WID * 2)) + 1;

              if MACRaw > GOB_WID then
                begin
                  MACRaw := MACRaw - GOB_WID;
                  MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                  GOB := (((MacRow - 1) div GOB_HIG) + 1) * 2;
                end

              else
                begin
                  MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                  GOB := ((((MacRow - 1) div GOB_HIG) + 1) * 2) - 1;
                end;
            end
```

```
            else
              begin
                TrRow := ((Count - 1) div (GOB_WID )) + 1;
                MACRaw := ((Count - 1) mod (GOB_WID)) + 1;

                CLine := ((TrRow - 1) mod (BLK_WID * 2)) + 1;
                CLine2 := ((TrRow - 1) mod (BLK_WID)) + 1;
                MACRow := ((TrRow - 1) div (BLK_WID * 2)) + 1;

                MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                GOB := (((MacRow - 1) div GOB_HIG) + 1);
              end;

            Pic[GOB]^.Macs[MAC].RawMacYArray[CLine] := InY[Idx2];

            Count := Count + 1;
          end;

      end;

{* This Fiddely Bit Serves to copy a 8 x 8 block into the raw Frame       *}
{* Because When its loaded the bytes are not arranged in blocks but in a  *}
{* Sequence, which gets placed in the load 8x8 block. Therefor this part  *}
{* is required to convert this block in to the large frame size.          *}


    Count := 1;
    for Idx1 := 1 to NumMACS do
      begin
        read(UFile,InU);
        read(VFile,InV);

        for Idx2 := 1 to (BLK_WID) do
          begin
            if MovType in [CIF,NTSC] then
              begin
                TrRow := ((Count - 1) div (GOB_WID * 2)) + 1;
                MACRaw := ((Count - 1) mod (GOB_WID * 2)) + 1;

                CLine := ((TrRow - 1) mod (BLK_WID)) + 1;
                CLine2 := ((TrRow - 1) mod (BLK_WID)) + 1;
                MACRow := ((TrRow - 1) div (BLK_WID)) + 1;

                if MACRaw > GOB_WID then
                  begin
                    MACRaw := MACRaw - GOB_WID;
                    MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                    GOB := (((MacRow - 1) div GOB_HIG) + 1) * 2;
                  end

                else
                  begin
                    MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                    GOB := ((((MacRow - 1) div GOB_HIG) + 1) * 2) - 1;
                  end;
              end
            else
              begin
```

```
            TrRow := ((Count - 1) div (GOB_WID )) + 1;
            MACRaw := ((Count - 1) mod (GOB_WID)) + 1;

            CLine := ((TrRow - 1) mod (BLK_WID)) + 1;
            CLine2 := ((TrRow - 1) mod (BLK_WID)) + 1;
            MACRow := ((TrRow - 1) div (BLK_WID)) + 1;

            MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
            GOB := (((MacRow - 1) div GOB_HIG) + 1);
          end;

        Pic[GOB]^.Macs[MAC].RawMacUArray[CLine] := InU[Idx2];
        Pic[GOB]^.Macs[MAC].RawMacVArray[CLine] := InV[Idx2];

        Count := Count + 1;
      end;

  end;
 close(YFile);
 close(UFile);
 close(VFile);
end;

procedure RawtPicCpy(var I : Infotype;Pic : Pictype; var RPic : Rpictype);
{* This Procedure Copies A Raw Picture Frame To A Integer Picture Frame    *}

var
 GOB,MAC,Idx1,Idx2 : word;
 Gob_Num : word;

begin
 Gob_Num := Fr_Chars(I.Mov_Type,'G');
 for GOB := 1 to Gob_Num do
  begin
   for Mac := 1 to MAC_GOB do
    begin
     for Idx1 := 1 to (BLK_WID * 2) do
      begin
       for Idx2 := 1 to (BLK_WID * 2) do
        begin
         Pic[GOB]^.Macs[MAC].MQuant := IMQUANT;
         Pic[GOB]^.Macs[MAC].MType := 1;
         Pic[GOB]^.Macs[MAC].RawMacYArray[Idx1,Idx2] :=
          RPic[GOB]^.Macs[MAC].RawMacYArray[Idx1,Idx2];
         If ((Idx1 mod 2 ) = 0) and ((Idx2 mod 2) = 0) then
          begin
           Pic[GOB]^.Macs[MAC].RawMacUArray[Idx1 div 2,Idx2 div 2] :=
           RPic[GOB]^.Macs[MAC].RawMacUArray[Idx1 div 2,Idx2 div 2];
           Pic[GOB]^.Macs[MAC].RawMacVArray[Idx1 div 2,Idx2 div 2] :=
           RPic[GOB]^.Macs[MAC].RawMacVArray[Idx1 div 2,Idx2 div 2];
          end;
        end;
      end;
    end;
  end;

end;
```

Appendix A

```
procedure ConvToM_G(Mov_Type : byte;RawMAC : word; var GOB,MAC : word);
{* Returns GOB and MAC of a given Raw Macroblock.                    *}

var
  Idx1 : word;
  MACRow,MACCol : word;

begin
  If Mov_Type in [NTSC,CIF] then
    begin
      MACRow := ((RawMAC - 1) div (GOB_WID * 2)) + 1;
      MACCol := ((RawMAC - 1) mod (GOB_WID * 2)) + 1;

      if MACCol > GOB_WID then
        begin
          MACCol := MACCol - GOB_WID;
          MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACCol;
          GOB := (((MacRow - 1) div GOB_HIG) + 1) * 2;
        end

      else
        begin
          MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACCol;
          GOB := ((((MacRow - 1) div GOB_HIG) + 1) * 2) - 1;
        end;
    end
  else
   If Mov_Type in [QCIF] then
     begin
       MACRow := ((RawMAC - 1) div (GOB_WID)) + 1;
       MACCol := ((RawMAC - 1) mod (GOB_WID)) + 1;
       MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACCol;
       GOB := (((MacRow - 1) div GOB_HIG) + 1);      { * 2) - 1;}
     end;

end;

procedure ConMCRData(Num : word; var MAC,Row,Col : byte; SBlk : boolean);
{* This Function Returns ther mac and a 2D Pointer to a value in the mac    *}

var
  Row1,Col1,MACRow,MACCol : word;
  BlkWid,BlkHig : word;

begin
  if SBlk then
    begin
      BlkWid := BLK_WID*2;
      BlkHig := BLK_HIG*2;
    end
  else
    begin
      BlkWid := BLK_WID;
      BlkHig := BLK_HIG;
    end;

  Row1 := (Num - 1) div (3 * BlkWid) + 1;
```

```
  Col1 := (Num - 1) mod (3 * BlkWid) + 1;
  MACRow := ((Row1 - 1) div BlkHig) + 1;
  MACCol := ((Col1 - 1) div BlkWid) + 1;
  Row := ((Row1 - 1) mod BlkHig) + 1;
  Col := ((Col1 - 1) mod BlkWid) + 1;

  MAC := (MacRow * 3) + MACCol - 3;        {* Hmmm Interesting Discovery *}

end;


function CBPCalc(var UMac : Macblkptrtype) : byte;
{* This Function Calculates the CBP of a Given Macroblk.              *}

var
  Row,Col : byte;
  Add1,Add2,Add3,Add4,Add5,Add6 : single;
  CBP : byte;

begin
  Add1 := 0;
  Add2 := 0;
  Add3 := 0;
  Add4 := 0;
  Add5 := 0;
  Add6 := 0;
  CBP := 0;

{* This Part Just Checks to see which Block in the MAC is zero. ie don't   *}
{* transmit. *}

  for Row := 1 to BLK_HIG do
    for Col := 1 to BLK_WID do
      begin
        Add1 := Add1 + abs(UMac^.RawMacYArray[Row,Col]);
        Add2 := Add2 + abs(UMac^.RawMacYArray[Row,Col+BLK_WID]);
        Add3 := Add3 + abs(UMac^.RawMacYArray[Row+BLK_HIG,Col]);
        Add4 := Add4 + abs(UMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]);

        Add5 := Add5 + abs(UMac^.RawMacUArray[Row,Col]);
        Add6 := Add6 + abs(UMac^.RawMacVArray[Row,Col]);
      end;

  if Add1 <> 0 then
    CBP := CBP + 32;
  if Add2 <> 0 then
    CBP := CBP + 16;
  if Add3 <> 0 then
    CBP := CBP + 8;
  if Add4 <> 0 then
    CBP := CBP + 4;
  if Add5 <> 0 then
    CBP := CBP + 2;
  if Add6 <> 0 then
    CBP := CBP + 1;

  CBPCalc := CBP;
end;
```

Appendix A

```
function MCSearch(var MCA : Mcatype; var UPic : Macblkptrtype; var CMac : RMacptrtype) : boolean;
{* This Function Does a full Slow motion Search with SRCH_WID.          *}

var
  Count1,Count2 : shortint;
  HStart,HEnd,VStart,VEnd : shortint;
  Pos1,Pos2,Pos3,Pos4 : word;
  MAC,Row,Col : byte;
  Diff1,Diff2,Diff3,Diff4,Min : longint;
  Idx1,Idx2 : byte;
  MVH,MVV : shortint;
  A,B,C,D : longint;
  Tmpblk : Macblkptrtype;
  MCVar,Mean1,Mean2,ORVar,Sum1,Sum2,Tmp : single;

begin
  MCSearch := False;
  HStart := -SRCH_WID;
  VStart := -SRCH_WID;
  HEnd := SRCH_WID;
  VEnd := SRCH_WID;
  new(Tmpblk);

  if MCA[4] = nil then
    HStart := 0
  else
    HStart := -SRCH_WID;

  if MCA[2] = nil then
    VStart := 0
  else
    VStart := -SRCH_WID;

  if MCA[6] = nil then
    HEnd := 0
  else
    HEnd := SRCH_WID;

  if MCA[8] = nil then
    VEnd := 0
  else
    VEnd := SRCH_WID;

  Min := MAXLONGINT;

  for Count1 := 0 to SRCH_WID do
    for Count2 := 0 to SRCH_WID do
      begin
        Diff1 := 0;
        Diff2 := 0;
        Diff3 := 0;
        Diff4 := 0;

        for Idx1 := 1 to (BLK_HIG) do
          for Idx2 := 1 to (BLK_WID) do
            begin
              if -Count1 < VStart then
```

Appendix A

```
  begin
    Diff1 := MAXLONGINT;
    Diff2 := MAXLONGINT;
  end
else
  begin
    if -Count2 < HStart then
      begin
        Diff1 := MAXLONGINT;
      end
    else
      begin
        Pos1 := ZRO_POS + -Count2 + (3 * BLK_WID * 2 * -Count1)
        + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1);
        ConMCRData(Pos1,MAC,Row,Col,True);
        A := CMac^.RawMacYArray[Idx1,Idx2] -
            MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + -Count2 + (3 * BLK_WID * 2 * -Count1)
        + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1+BLK_WID);
        ConMCRData(Pos1,MAC,Row,Col,True);
        B := CMac^.RawMacYArray[Idx1,Idx2+BLK_WID] -
            MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + -Count2 + (3 * BLK_WID * 2 * -Count1)
        + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1);
        ConMCRData(Pos1,MAC,Row,Col,True);
        C := CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2] -
            MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + -Count2 + (3 * BLK_WID * 2 * -Count1)
        + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1+BLK_WID);
        ConMCRData(Pos1,MAC,Row,Col,True);
        D := CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2+BLK_WID] -
            MCA[MAC]^.RawMacYArray[Row,Col];
        Diff1 := Diff1 + abs(A) + abs(B) + abs(C) + abs(D);
      end;
    if Count2 > HEnd then
      begin
        Diff2 := MAXLONGINT;
      end
    else
       begin
         Pos1 := ZRO_POS + Count2 + (3 * BLK_WID * 2 * -Count1)
        + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1);
        ConMCRData(Pos1,MAC,Row,Col,True);
        A := CMac^.RawMacYArray[Idx1,Idx2] -
            MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + Count2 + (3 * BLK_WID * 2 * -Count1)
        + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1+BLK_WID);
        ConMCRData(Pos1,MAC,Row,Col,True);
        B := CMac^.RawMacYArray[Idx1,Idx2+BLK_WID] -
            MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + Count2 + (3 * BLK_WID * 2 * -Count1)
        + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1);
        ConMCRData(Pos1,MAC,Row,Col,True);
```

```
        C := CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2] -
          MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + Count2 + (3 * BLK_WID * 2 * -Count1)
        + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1+BLK_WID);
        ConMCRData(Pos1,MAC,Row,Col,True);
        D := CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2+BLK_WID] -
          MCA[MAC]^.RawMacYArray[Row,Col];
        Diff2 := Diff2 + abs(A) + abs(B) + abs(C) + abs(D);
       end;
  end;
if Count1 > VEnd then
  begin
    Diff3 := MAXLONGINT;
    Diff4 := MAXLONGINT;
  end
else
  begin
    if -Count2 < HStart then
      begin
        Diff3 := MAXLONGINT;
      end
    else
      begin
        Pos1 := ZRO_POS + -Count2 + (3 * BLK_WID * 2 * Count1)
        + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1);
        ConMCRData(Pos1,MAC,Row,Col,True);
        A := CMac^.RawMacYArray[Idx1,Idx2] -
          MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + -Count2 + (3 * BLK_WID * 2 * Count1)
        + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1+BLK_WID);
        ConMCRData(Pos1,MAC,Row,Col,True);
        B := CMac^.RawMacYArray[Idx1,Idx2+BLK_WID] -
          MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + -Count2 + (3 * BLK_WID * 2 * Count1)
        + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1);
        ConMCRData(Pos1,MAC,Row,Col,True);
        C := CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2] -
          MCA[MAC]^.RawMacYArray[Row,Col];

        Pos1 := ZRO_POS + -Count2 + (3 * BLK_WID * 2 * Count1)
        + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1+BLK_WID);
        ConMCRData(Pos1,MAC,Row,Col,True);
        D := CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2+BLK_WID] -
          MCA[MAC]^.RawMacYArray[Row,Col];
        Diff3 := Diff3 + abs(A) + abs(B) + abs(C) + abs(D);
      end;

    if Count2 > HEnd then
      begin
        Diff4 := MAXLONGINT;
      end
    else
      begin
        Pos1 := ZRO_POS + Count2 + (3 * BLK_WID * 2 * Count1)
        + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1);
```

```
                    ConMCRData(Pos1,MAC,Row,Col,True);
                    A := CMac^.RawMacYArray[Idx1,Idx2] -
                       MCA[MAC]^.RawMacYArray[Row,Col];

                    Pos1 := ZRO_POS + Count2 + (3 * BLK_WID * 2 * Count1)
                    + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1+BLK_WID);
                    ConMCRData(Pos1,MAC,Row,Col,True);
                    B := CMac^.RawMacYArray[Idx1,Idx2+BLK_WID] -
                       MCA[MAC]^.RawMacYArray[Row,Col];

                    Pos1 := ZRO_POS + Count2 + (3 * BLK_WID * 2 * Count1)
                    + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1);
                    ConMCRData(Pos1,MAC,Row,Col,True);
                    C := CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2] -
                       MCA[MAC]^.RawMacYArray[Row,Col];

                    Pos1 := ZRO_POS + Count2 + (3 * BLK_WID * 2 * Count1)
                    + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1+BLK_WID);
                    ConMCRData(Pos1,MAC,Row,Col,True);
                    D := CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2+BLK_WID] -
                       MCA[MAC]^.RawMacYArray[Row,Col];
                    Diff4 := Diff4 + abs(A) + abs(B) + abs(C) + abs(D);
                  end;
              end;


          end;

{* This Difference is the Absolute difference between previous and current  *}
{* blocks.                                                                  *}
      if Diff1 < Min then
        begin
          Min := Diff1;
          MVH := -Count2;
          MVV := -Count1;
        end;

      if Diff2 < Min then
        begin
          Min := Diff2;
          MVH := Count2;
          MVV := -Count1;
        end;

      if Diff3 < Min then
        begin
          Min := Diff2;
          MVH := -Count2;
          MVV := Count1;
        end;

      if Diff4 < Min then
        begin
          Min := Diff2;
          MVH := Count2;
          MVV := Count1;
        end;
    end;
```

```
for Idx1 := 1 to BLK_HIG do
  for Idx2 := 1 to BLK_WID do
    begin
      Pos1 := ZRO_POS + MVH + (3 * BLK_WID * 2 * MVV)
      + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1);
      ConMCRData(Pos1,MAC,Row,Col,True);

      UPic^.RawMacYArray[Idx1,Idx2] := CMac^.RawMacYArray[Idx1,Idx2] -
                          MCA[MAC]^.RawMacYArray[Row,Col];

      Pos1 := ZRO_POS + MVH + (3 * BLK_WID * 2 * MVV)
      + (BLK_WID * 2 * (Idx1-1) * 3) + (Idx2-1+BLK_WID);
      ConMCRData(Pos1,MAC,Row,Col,True);

      UPic^.RawMacYArray[Idx1,Idx2+BLK_WID] :=
        CMac^.RawMacYArray[Idx1,Idx2+BLK_WID] -
        MCA[MAC]^.RawMacYArray[Row,Col];

      Pos1 := ZRO_POS + MVH + (3 * BLK_WID * 2 * MVV)
      + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1);
      ConMCRData(Pos1,MAC,Row,Col,True);

      UPic^.RawMacYArray[Idx1+BLK_HIG,Idx2] :=
        CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2] -
        MCA[MAC]^.RawMacYArray[Row,Col];

      Pos1 := ZRO_POS + MVH + (3 * BLK_WID * 2 * MVV)
      + (BLK_WID * 2 * (Idx1-1+BLK_HIG) * 3) + (Idx2-1+BLK_WID);
      ConMCRData(Pos1,MAC,Row,Col,True);

      UPic^.RawMacYArray[Idx1+BLK_HIG,Idx2+BLK_WID] :=
        CMac^.RawMacYArray[Idx1+BLK_HIG,Idx2+BLK_WID] -
        MCA[MAC]^.RawMacYArray[Row,Col];

      Pos1 := ZRO_POS1 + round(MVH/2) + (3 * BLK_WID * round(MVV/2))
      + (BLK_WID * (Idx1-1) * 3) + (Idx2-1);
      ConMCRData(Pos1,MAC,Row,Col,False);

      UPic^.RawMacUArray[Idx1,Idx2] :=
        CMac^.RawMacUArray[Idx1,Idx2] -
        MCA[MAC]^.RawMacUArray[Row,Col];

      UPic^.RawMacVArray[Idx1,Idx2] :=
        CMac^.RawMacVArray[Idx1,Idx2] -
        MCA[MAC]^.RawMacVArray[Row,Col];
    end;


  UPic^.MQuant := IMQUANT;
{* If Its not a direct copy (calculated here) then do the else bit.      *}

  if (MVH = 0) and (MVV = 0) then
    begin
      if Min = 0 then
        begin
          UPic^.MType := 5;
          UPic^.MVV := MVV;
```

```
          UPic^.MVH := MVH;
        end
      else
        begin
          UPic^.MType := 3;
          UPic^.CBP := 63;
        end;
    end
  else
    begin
{* This Bit deals with the varience Of the Block and the Original Block to *}
{* determine if to transmit a MC Frame Or an Intra Frame Or A Inter Frame  *}
      Sum1 := 0;
      Mean1 := 0;
      Sum2 := 0;
      Mean2 := 0;

      for Idx1 := 1 to (BLK_HIG * 2) do
        for Idx2 := 1 to (BLK_WID * 2) do
          begin
            Tmp := UPic^.RawMacYArray[Idx1,Idx2];
            Sum1 := Sum1 + (Tmp * UPic^.RawMacYArray[Idx1,Idx2]);
            Mean1 := Mean1 + UPic^.RawMacYArray[Idx1,Idx2];
            Tmp := CMac^.RawMacYArray[Idx1,Idx2];
            Sum2 := Sum2 + (Tmp * CMac^.RawMacYArray[Idx1,Idx2]);
            Mean2 := Mean2 + CMac^.RawMacYArray[Idx1,Idx2];
          end;

      Sum1 := Sum1 / (BLK_WID * 2 * BLK_HIG * 2);
      Mean1 := Mean1 / (BLK_WID * 2 * BLK_HIG * 2);
      Sum2 := Sum2 / (BLK_WID * 2 * BLK_HIG * 2);
      Mean2 := Mean2 / (BLK_WID * 2 * BLK_HIG * 2);

      MCVar := Sum1 - (Mean1 * Mean1);
      ORVar := Sum2 - (Mean2 * Mean2);

      If MCVar < ORVar then
        begin
          if Min = 0 then
            begin
              UPic^.MType := 5;
              UPic^.MVV := MVV;
              UPic^.MVH := MVH;
            end
          else
            begin
              UPic^.MType := 6;
              UPic^.MVV := MVV;
              UPic^.MVH := MVH;
              UPic^.CBP := 63;
            end;
        end
      else
        begin
          MCSearch := True;
          UPic^.MType := 1;
          UPic^.MQuant := IMQUANT;
          for Idx1 := 1 to (BLK_HIG * 2) do
```

```
            for Idx2 := 1 to (BLK_WID * 2) do
              begin
                UPic^.RawMacyArray[Idx1,Idx2] := CMac^.RawMacyArray[Idx1,Idx2];
                if ((Idx1 mod 2) = 0) and ((Idx2 mod 2) = 0) then
                  begin
                    UPic^.RawMacUArray[Idx1 div 2,Idx2 div 2] :=
                      CMac^.RawMacUArray[Idx1 div 2,Idx2 div 2];
                    UPic^.RawMacVArray[Idx1 div 2,Idx2 div 2] :=
                      CMac^.RawMacVArray[Idx1 div 2,Idx2 div 2];
                  end;
              end;
        end;
    end;
  dispose(TmpBlk);
end;


procedure MotionAnalysis (var I : Infotype; var UPic : Pictype; CPic, PPic : Rpictype; Pfe : boolean);
{* This Procedure Performs All the Motion Analysis, Compensation and      *}
{*                                          estimation *}

var
  Idx1,Idx2 : word;
  GOB,MAC,GOB1,MAC1,MAC_Num : word;
  MCA : Mcatype;
  Top,Bottom,Left,Right : boolean;
  UMac : Macblkptrtype;
  Intra : boolean;
  CMac : Rmacptrtype;


begin
  if not (I.Mov_Type in [CIF,QCIF,NTSC]) then
    Die ('Illigal Picture type (MotionAnalysis)',3);

  if Not Pfe then                        {* if No Previous Picture *}
    begin
      RawtPicCpy(I,UPic,CPic);
    end

  else
    begin
      MAC_Num := Fr_Chars(I.Mov_Type,'M');
      for Idx1 := 1 to MAC_Num do
        begin
          write('>');
          ConvToM_G(I.Mov_Type,Idx1,GOB,MAC);
          UMac := @((UPic[GOB])^.Macs[MAC]);
          CMac := @((CPic[GOB])^.Macs[MAC]);

{* All this Code Does is to detremine the surrounding Blocks for a Macro-   *}
{* Block in a particular Frame Type.                                        *}

          if I.Mov_Type = CIF then
            begin
              If (GOB mod 2) = 1 then
                begin
                  if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
```

```
            begin
              Left := False;
            end
          else
            begin
              Left := True;
            end;
        end
      else
        begin
          if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
            begin
              Right := False;
            end
          else
            begin
              Right := True;
            end;
        end;

      If GOB in [1,2] then
        begin
          if MAC in [1..GOB_WID] then
            begin
              Top := False;
            end
          else
            begin
              Top := True;
            end;
        end;

      If GOB in [11,12] then
        begin
          if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
            begin
              Bottom := False;
            end
          else
            begin
              Bottom := True;
            end;
        end;
      if not (GOB in [1,2,11,12]) then
        begin
          Top := True;
          Bottom := True;
        end;
    end;

  if I.Mov_Type = QCIF then
    begin
      if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
        begin
          Left := False;
        end
      else
        begin
```

```
        Left := True;
      end;

    if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
      begin
        Right := False;
      end
    else
      begin
        Right := True;
      end;

    If GOB = 1 then
      begin
        if MAC in [1..GOB_WID] then
          begin
            Top := False;
          end
        else
          begin
            Top := True;
          end;
      end;

    If GOB = 3 then
      begin
        if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
          begin
            Bottom := False;
          end
        else
          begin
            Bottom := True;
          end;
      end;

    if not (GOB in [1,3]) then
      begin
        Top := True;
        Bottom := True;
      end;
  end;

if I.Mov_Type = NTSC then
  begin
    Left := True;
    Right := True;
    Top := True;
    Bottom := True;
    If (GOB mod 2) = 1 then
      begin
        if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
          begin
            Left := False;
          end
        else
          begin
            Left := True;
```

```
            end;
        end
      else
        begin
          if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
            begin
              Right := False;
            end
          else
            begin
              Right := True;
            end;
        end;

      If GOB in [1,2] then
        begin
          if MAC in [1..GOB_WID] then
            begin
              Top := False;
            end
          else
            begin
              Top := True;
            end;
        end;

      If GOB in [9,10] then
        begin
          if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
            begin
              Bottom := False;
            end
          else
            begin
              Bottom := True;
            end;
        end;

      if not (GOB in [1,2,9,10]) then
        begin
          Top := True;
          Bottom := True;
        end;
    end;

  if Top then
    begin
      ConvToM_G(I.Mov_Type,(Idx1 - GOB_WID*2),GOB1,MAC1);
      MCA[2] := @((PPic[GOB1])^.Macs[MAC1]);
    end
  else
    MCA[2] := nil;

  if Top and Right then
    begin
      ConvToM_G(I.Mov_Type,(Idx1 - GOB_WID*2 + 1),GOB1,MAC1);
      MCA[3] := @((PPic[GOB1])^.Macs[MAC1]);
    end
```

```
            else
              MCA[3] := nil;

            if Right then
              begin
                ConvToM_G(I.Mov_Type,(Idx1 + 1),GOB1,MAC1);
                MCA[6] := @((PPic[GOB1])^.Macs[MAC1]);
              end
            else
              MCA[6] := nil;

            if Right and Bottom then
              begin
                ConvToM_G(I.Mov_Type,(Idx1 + 1 + GOB_WID*2),GOB1,MAC1);
                MCA[9] := @((PPic[GOB1])^.Macs[MAC1]);
              end
            else
              MCA[9] := nil;

            if Bottom then
              begin
                ConvToM_G(I.Mov_Type,(Idx1 + GOB_WID*2),GOB1,MAC1);
                MCA[8] := @((PPic[GOB1])^.Macs[MAC1]);
              end
            else
              MCA[8] := nil;

            if Bottom and Left then
              begin
                ConvToM_G(I.Mov_Type,(Idx1 + GOB_WID*2 - 1),GOB1,MAC1);
                MCA[7] := @((PPic[GOB1])^.Macs[MAC1]);
              end
            else
              MCA[7] := nil;

            if Left then
              begin
                ConvToM_G(I.Mov_Type,(Idx1 - 1),GOB1,MAC1);
                MCA[4] := @((PPic[GOB1])^.Macs[MAC1]);
              end
            else
              MCA[4] := nil;

            if Left and Top then
              begin
                ConvToM_G(I.Mov_Type,(Idx1 - 1 - GOB_WID*2),GOB1,MAC1);
                MCA[1] := @((PPic[GOB1])^.Macs[MAC1]);
              end
            else
              MCA[1] := nil;

            MCA[5] := @((PPic[GOB])^.Macs[MAC]);
            Intra := MCSearch(MCA,UMac,CMac);

          end;
        end;
    end;
```

```
procedure WriteBS(var BS : Bstype; InStr : string);
{* This Procedure Writes InStr to the bitsream. Gets Called A # Of times.  *}

var
  Idx1,Idx2 : word;
  TmpStr : string;
  OutVal : byte;

begin
  BS.Tr_Bits := BS.Tr_Bits + InStr;
  BS.Vbits := length(BS.Tr_Bits);

{ if INFO.DEBUGVAL = 1 then
    begin
      write(' Out:',InStr);
      if readkey = ' ' then halt(1);
    end;}


  while (BS.Vbits) > 8 do
    begin
      TmpStr := '';
      for Idx1 := 1 to 8 do
        TmpStr := TmpStr + BS.Tr_Bits[Idx1];
      OutVal := StB(TmpStr);
      write(BS.BsFile,Outval);
      TmpStr := '';
      For Idx1 := 9 to BS.Vbits do
        TmpStr := TmpStr + BS.Tr_Bits[Idx1];
      BS.Tr_Bits := TmpStr;
      BS.VBits := length(BS.Tr_Bits);
    end;
end;


procedure WritePicHeader(var I : Infotype; var BS : Bstype; Tr : word);
{* This Procedure Writes The Picture header To the Bitstream.          *}

var
  TStr1,TStr2 : String;
  Idx1 : word;
  Tr1 : byte;

begin
  WriteBS(BS,'00000000000000010000');                    {* PSC *}

  Tr1 := (Tr mod 32);
  TStr1 :='';
  TStr2 := '';
  TStr1 := BtS(Tr1);
  for Idx1 := 4 to 8 do
    TStr2 := TStr2 + TStr1[Idx1];

  WriteBS(BS,TStr2);                        {* Temporal Reference *}

  WriteBS(BS,'000');                {* Pic Freeze, split Screen etc. *}

  if I.Mov_Type in [CIF,NTSC] then                    {* Pic Type *}
```

```
    WriteBS(BS,'1')
  else
    WriteBS(BS,'0');

  WriteBS(BS,'00');                          {* Reserved *}

  if I.Mov_Type = NTSC then                  {* Pic type Extention for NTSC *}
    begin
      WriteBS(BS,'1');                       {* Spare Enable *}
      WriteBS(BS,'10001100');                    {* NTSC Frame *}
      WriteBS(BS,'0');                       {* Spare Disable *}
    end
  else
    WriteBS(BS,'0');
end;

procedure WriteGOBHeader(var I : Infotype; var BS : Bstype; GOB,GQuant : byte);
{* This Procedure Writes The Picture header To the Bitstream.          *}

var
  TStr1,TStr2 : String;
  Idx1 : word;
  MVDH,MVDV : shortint;

begin
  WriteBS(BS,'0000000000000001');                {* GSC *}
  TStr1 := BtS(GOB);
  TStr2 := '';
  for Idx1 := 5 to 8 do
    TStr2 := TStr2 + TStr1[Idx1];

  WriteBS(BS,TStr2);                         {* GOB # *}

  TStr1 := BtS(GQuant);
  TStr2 := '';

  for Idx1 := 4 to 8 do
    TStr2 := TStr2 + TStr1[Idx1];

  WriteBS(BS,TStr2);                             {* GQuant *}
  WriteBS(BS,'0');                           {* Spare Enable *}
end;

function Quantise(Value : DCTvaltype; Quant : byte) : Picvaltype;
{* This Function CCITT Quantises a value with a given Quant.          *}

var
  Tmp : Picvaltype;

begin
  if Value = 0 then
    Quantise := 0
  else
    begin
      if (Quant mod 2) = 1 then
        begin
          Tmp := round(Value/(2*Quant));
        end
```

```
        else
          begin
            if Value > 0 then
              Tmp := round((Value+1)/(2*Quant))
            else
              Tmp := round((Value-1)/(2*Quant));
          end;

        if Tmp > 127 then
          Tmp := 127;

        if Tmp < -127 then
          Tmp := -127;
        Quantise := Tmp;
      end;
end;

procedure DCT_Quant(var UMac : MacBlkptrtype; Quant : byte);
{* This Procedure Does the DCT for the entire macroblock and calls        *}
{* Quantisation For each value.                                           *}

var
  Mult,Mult2,Calc : DCTvaltype;
  jSum1,iSum1 : DCTvaltype;
  jSum2,iSum2 : DCTvaltype;
  jSum3,iSum3 : DCTvaltype;
  jSum4,iSum4 : DCTvaltype;
  jSum5,iSum5 : DCTvaltype;
  jSum6,iSum6 : DCTvaltype;
  u,v,i,j,DSiz : byte;
  NMac : MacBlkptrtype;

begin
  Mult := 1/sqrt(2);
  Mult2 := 1/4;
  DSiz := BLK_WID + BLK_HIG;
  new (NMac);
  with UMac^ do
  for u := 0 to (BLK_HIG - 1) do
    begin
      for v := 0 to (BLK_WID - 1) do
        begin
          iSum1 := 0;
          iSum2 := 0;
          iSum3 := 0;
          iSum4 := 0;
          iSum5 := 0;
          iSum6 := 0;
          for i := 0 to (BLK_HIG - 1) do
            begin
              jSum1 := 0;
              jSum2 := 0;
              jSum3 := 0;
              jSum4 := 0;
              jSum5 := 0;
              jSum6 := 0;

              for j := 0 to (BLK_WID - 1) do
```

```
        begin
          Calc := cos((2*i + 1)*u*pi/DSiz) *
                cos((2*j + 1)*v*pi/DSiz);
                      {* saving As calculateted once *}

          jSum1 := jSum1 + (RawMacYArray[(i+1),(j+1)] * Calc);
          jSum2 := jSum2 + (RawMacYArray[(i+1),(j+1+BLK_WID)] * Calc);
          jSum3 := jSum3 + (RawMacYArray[(i+1+BLK_HIG),(j+1)] * Calc);
          jSum4 := jSum4 + (RawMacYArray[(i+1+BLK_HIG),(j+1+BLK_WID)] * Calc);

          jSum5 := jSum5 + (RawMacUArray[(i+1),(j+1)] * Calc);
          jSum6 := jSum6 + (RawMacVArray[(i+1),(j+1)] * Calc);
        end;

      iSum1 := iSum1 + jSum1;
      iSum2 := iSum2 + jSum2;
      iSum3 := iSum3 + jSum3;
      iSum4 := iSum4 + jSum4;
      iSum5 := iSum5 + jSum5;
      iSum6 := iSum6 + jSum6;

    end;

  if u = 0 then
    begin
      iSum1 := iSum1 * Mult;
      iSum2 := iSum2 * Mult;
      iSum3 := iSum3 * Mult;
      iSum4 := iSum4 * Mult;
      iSum5 := iSum5 * Mult;
      iSum6 := iSum6 * Mult;
    end;

  if v = 0 then
    begin
      iSum1 := iSum1 * Mult;
      iSum2 := iSum2 * Mult;
      iSum3 := iSum3 * Mult;
      iSum4 := iSum4 * Mult;
      iSum5 := iSum5 * Mult;
      iSum6 := iSum6 * Mult;
    end;

  iSum1 := iSum1 * Mult2;
  iSum2 := iSum2 * Mult2;
  iSum3 := iSum3 * Mult2;
  iSum4 := iSum4 * Mult2;
  iSum5 := iSum5 * Mult2;
  iSum6 := iSum6 * Mult2;

  NMac^.RawMacYArray[u+1,v+1] := Quantise(iSum1,Quant);
  NMac^.RawMacYArray[u+1,v+1+BLK_WID] := Quantise(iSum2,Quant);
  NMac^.RawMacYArray[u+1+BLK_HIG,v+1] := Quantise(iSum3,Quant);
  NMac^.RawMacYArray[u+1+BLK_HIG,v+1+BLK_WID] := Quantise(iSum4,Quant);
  NMac^.RawMacUArray[u+1,v+1] := Quantise(iSum5,Quant);
  NMac^.RawMacVArray[u+1,v+1] := Quantise(iSum6,Quant);
```

```
          end;
      end;

    UMac^.RawMacYArray := NMac^.RawMacYArray;
    UMac^.RawMacUArray := NMac^.RawMacUArray;
    UMac^.RawMacVArray := NMac^.RawMacVArray;
    dispose(NMac);
end;


procedure SignSave(var BS : Bstype; Value : integer);
{* This Procedure Saves data that is not run length, using signs.          *}

var
  Idx1 : word;
  TStr1,TStr2 : string;
  Inbyte : byte;

begin
  if Value < 0 then
    WriteBS(BS,'1')
  else
    WriteBS(BS,'0');

  InByte := abs(Value);
  TStr1 := BtS(InByte);
  TStr2 := '';
  for Idx1 := 2 to 8 do
   TStr2 := TStr2 + TStr1[Idx1];
  WriteBS(BS,TStr2);

end;


function GetCoffStr(var I : Infotype; Run,Level : byte) : string;
{* This Function Returns The VLC corresponding to the Run and Level        *}

var
  Idx1 : word;
  Match : boolean;

begin
  Match := False;
  Idx1 := 2;
  while (not Match) and (Idx1 < 64) do
    begin
      if (I.CoTab[Idx1].Run = Run) and (I.CoTab[Idx1].Level = Level) then
        begin
          GetCoffStr := I.CoTab[Idx1].COFF;
          Match := True;
        end
      else
        Idx1 := Idx1 + 1;
    end;
  if not Match then
    GetCoffStr := 'Escape';

end;
```

Appendix A

```
procedure BlkSave(var I : Infotype; var BS : Bstype; var Ln : Blklinetype; ASave : boolean);
{* This Procedure Saves An Entire Block( that Selected to save) to the     *}
{* bitstream.                                                              *}

var
  Run,Count : byte;
  Level : integer;
  Idx1 : word;
  Sign : shortint;
  WStr,TStr1,TStr2 : string;
  Fst : boolean;


begin
  if ASave then
    begin
      Run := 0;
      Fst := True;
      for Idx1 := 1 to (BLK_WID*BLK_HIG) do
        begin
          if Ln[Idx1] = 0 then
            begin
              Run := Run + 1;
            end
          else
            begin
              Level := abs(Ln[Idx1]);
              WStr := '';
              if (Run in [0..26]) and (Level in [1..15]) then
                begin
                  if (Run = 0) and (Level = 1) then
                    if Fst then
                      begin
                        WStr := '1';
                        Fst := False;
                      end
                    else
                      WStr := '11'
                  else
                    WStr := GetCoffStr(I,Run,Level);

                  if WStr <> 'Escape' then
                    begin
                      Fst := False;
                      WriteBS(BS,WStr);
                      if Ln[Idx1] < 0 then
                        WriteBS(BS,'1')
                      else
                        WriteBS(BS,'0');

                    end;
                end;

              if (not ((Run in [0..26]) and (Level in [1..15]))) or (WStr = 'Escape') then
                begin
                  FSt := False;
```

```
                WriteBS(BS,I.CoTab[1].COFF);
                TStr1 := BtS(Run);
                TStr2 := '';
                for Count := 3 to 8 do
                  TStr2 := TStr2 + TStr1[Count];
                WriteBS(BS,TStr2);
                SignSave(BS,Ln[Idx1]);
              end;
            Run := 0;
          end;

      end;
  end
else
  begin
    Run := 0;
    for Idx1 := 2 to (BLK_WID*BLK_HIG) do
      begin
        if Ln[Idx1] = 0 then
          begin
            Run := Run + 1;
          end
        else
          begin
            Level := abs(Ln[Idx1]);
            WStr := '';

            if (Run in [0..26]) and (Level in [1..15]) then
              begin
                if (Run = 0) and (Level = 1) then
                  WStr := '11'
                else
                  WStr := GetCoffStr(I,Run,Level);

                if WStr <> 'Escape' then
                  begin
                    WriteBS(BS,WStr);
                    if Ln[Idx1] < 0 then
                      WriteBS(BS,'1')
                    else
                      WriteBS(BS,'0');
                  end;
              end;

            if (not ((Run in [0..26]) and (Level in [1..15]))) or (WStr = 'Escape') then
              begin
                WriteBS(BS,I.CoTab[1].COFF);
                TStr1 := BtS(Run);
                TStr2 := '';
                for Count := 3 to 8 do
                  TStr2 := TStr2 + TStr1[Count];
                WriteBS(BS,TStr2);
                SignSave(BS,Ln[Idx1]);
              end;
            Run := 0;
          end;
      end;
  end;
```

```
end;


procedure SaveCofs(var I : Infotype; var BS : Bstype; var UMac : Macblkptrtype);
{* This Procedure selects the coefficients to save and calls Saveblk or    *}
{* sign save.                                                              *}

var
  Idx1 : word;
  Row,Col : byte;
  Ln1,Ln2,Ln3,Ln4 : Blklinetype;
  CBP : byte;

begin
  if UMac^.MType in [1,2] then
    begin
      write('.');
      for Idx1 := 2 to BLK_WID*BLK_HIG do
        begin
          Row := I.ZZRow[Idx1-1] + 1;
          Col := I.ZZCol[Idx1-1] + 1;
          Ln1[Idx1] := UMac^.RawMacYArray[Row,Col];
          Ln2[Idx1] := UMac^.RawMacYArray[Row,Col+BLK_WID];
          Ln3[Idx1] := UMac^.RawMacYArray[Row+BLK_HIG,Col];
          Ln4[Idx1] := UMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID];
        end;

      SignSave(BS,UMac^.RawMacYArray[1,1]);
      BlkSave(I,BS,Ln1,False);
      WriteBS(BS,'10');

      SignSave(BS,UMac^.RawMacYArray[1,1+BLK_WID]);
      BlkSave(I,BS,Ln2,False);
      WriteBS(BS,'10');

      SignSave(BS,UMac^.RawMacYArray[1+BLK_HIG,1]);
      BlkSave(I,BS,Ln3,False);
      WriteBS(BS,'10');

      SignSave(BS,UMac^.RawMacYArray[1+BLK_HIG,1+BLK_WID]);
      BlkSave(I,BS,Ln4,False);
      WriteBS(BS,'10');


      for Idx1 := 2 to BLK_WID*BLK_HIG do
        begin
          Row := I.ZZRow[Idx1-1] + 1;
          Col := I.ZZCol[Idx1-1] + 1;
          Ln1[Idx1] := UMac^.RawMacUArray[Row,Col];
          Ln2[Idx1] := UMac^.RawMacVArray[Row,Col];
        end;

      Ln1[1] := UMac^.RawMacUArray[1,1];
      Ln2[1] := UMac^.RawMacVArray[1,1];

      SignSave(BS,UMac^.RawMacUArray[1,1]);
```

```
        BlkSave(I,BS,Ln1,False);
        WriteBS(BS,'10');

        SignSave(BS,UMac^.RawMacVArray[1,1]);
        BlkSave(I,BS,Ln2,False);
        WriteBS(BS,'10');

if INFO.DEBUGVAL = 1 then
  begin
    writeln;
    write('Been Here');
  end;
    end;


  if UMac^.MType in [3,4,6,7,9,10] then
    begin
      write('#');
      CBP := UMac^.CBP;

      Ln1[1] := UMac^.RawMacYArray[1,1];
      Ln2[1] := UMac^.RawMacYArray[1,1+BLK_WID];
      Ln3[1] := UMac^.RawMacYArray[1+BLK_HIG,1];
      Ln4[1] := UMac^.RawMacYArray[1+BLK_HIG,1+BLK_WID];

      for Idx1 := 2 to BLK_WID*BLK_HIG do
        begin
          Row := I.ZZRow[Idx1-1] + 1;
          Col := I.ZZCol[Idx1-1] + 1;
          Ln1[Idx1] := UMac^.RawMacYArray[Row,Col];
          Ln2[Idx1] := UMac^.RawMacYArray[Row,Col+BLK_WID];
          Ln3[Idx1] := UMac^.RawMacYArray[Row+BLK_HIG,Col];
          Ln4[Idx1] := UMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID];
        end;

if INFO.DEBUGVAL = 1 then
  begin
    writeln;
    write('CBP:',UMac^.CBP,' CC:',CBPCalc(UMac));
  end;

{if INFO.DEBUGVAL = 1 then
  begin
    writeln;
    for Idx1 := 1 to 64 do
      write(Ln1[Idx1],',');
  end;}

      if (UMac^.CBP and 32) = 32 then
        begin
          BlkSave(I,BS,Ln1,True);
          WriteBS(BS,'10');
{if INFO.DEBUGVAL = 1 then
  write(' Y1');}

        end;

{if INFO.DEBUGVAL = 1 then
```

```
  begin
    writeln;
    for Idx1 := 1 to 64 do
      write(In2[Idx1],',');
  end;}

      if (UMac^.CBP and 16) = 16 then
        begin
          BlkSave(I,BS,Ln2,True);
          WriteBS(BS,'10');
{if INFO.DEBUGVAL = 1 then
  write(' Y2');}

        end;
{if INFO.DEBUGVAL = 1 then
  begin
    writeln;
    for Idx1 := 1 to 64 do
      write(In3[Idx1],',');
  end;}

      if (UMac^.CBP and 8) = 8 then
        begin
          BlkSave(I,BS,Ln3,True);
          WriteBS(BS,'10');
{if INFO.DEBUGVAL = 1 then
  write(' Y3');}

        end;
{if INFO.DEBUGVAL = 1 then
  begin
    writeln;
    for Idx1 := 1 to 64 do
      write(In4[Idx1],',');
  end;}

      if (UMac^.CBP and 4) = 4 then
        begin
          BlkSave(I,BS,Ln4,True);
          WriteBS(BS,'10');
{if INFO.DEBUGVAL = 1 then
  write(' Y4');}

        end;

      for Idx1 := 2 to BLK_WID*BLK_HIG do
        begin
          Row := I.ZZRow[Idx1-1] + 1;
          Col := I.ZZCol[Idx1-1] + 1;
          Ln1[Idx1] := UMac^.RawMacUArray[Row,Col];
          Ln2[Idx1] := UMac^.RawMacVArray[Row,Col];
        end;

      Ln1[1] := UMac^.RawMacUArray[1,1];
      Ln2[1] := UMac^.RawMacVArray[1,1];

{if INFO.DEBUGVAL = 1 then
  begin
```

```
    writeln;
    for Idx1 := 1 to 64 do
      write(ln1[Idx1],',');
  end;}
      if (UMac^.CBP and 2) = 2 then
        begin
          BlkSave(I,BS,Ln1,True);
          WriteBS(BS,'10');
{if INFO.DEBUGVAL = 1 then
  write(' U');}

        end;

{if INFO.DEBUGVAL = 1 then
  begin
    writeln;
    for Idx1 := 1 to 64 do
      write(ln2[Idx1],',');
    readln;
  end;    }

      if (UMac^.CBP and 1) = 1 then
        begin
          BlkSave(I,BS,Ln2,True);
          WriteBS(BS,'10');
{if INFO.DEBUGVAL = 1 then
  write(' V');}
        end;

{if INFO.DEBUGVAL = 1 then
  begin
    write(' Mac End:');
    readln;
  end;}

    end;

end;


procedure WriteMAC(var I : Infotype; var BS : Bstype; MAC : byte; var CMac,PMac : Macblkptrtype;
Quant : byte);
{* This Procedure Writes The macro Block Info To the Bitstream.          *}

var
  TStr1,TStr2 : String;
  Idx1 : word;
  MVDH,MVDV : shortint;
  CBP : byte;


begin
  if (MAC > 20) and INFO.DEBUG then
    INFO.DEBUGVAL := 1
  else
    INFO.DEBUGVAL := 0;

  WriteBS(BS,I.MBATab[1]);                    {* Mac Number *}
```

```
  if CMac^.MType in [1..4,6,7,9,10] then
    begin
      DCT_Qu..nt(CMac,Quant);     {* Perform DCT and Quantisation per MAC *}
    end
  else
    begin
            {* In Case OF MC with zero Displacement then Do Nothing *}
    end;

  if CMac^.MType in [2,4,7,10] then
    Idx1 := 1;
                    {* Do MQuant Not Used, For later Improvement *}

  if CMac^.MType in [3,4,6,7,9,10] then
    begin
      CBP := CBPCalc(CMac);
{if INFO.DEBUGVAL = 1 then
      write(' CBP');  }
      if CBP = 0 then
        begin
          CMac^.MType := 5;
          CMac^.MVH := 0;
          CMac^.MVV := 0;
        end
      else
        begin
          CMac^.CBP := CBP;
        end;
    end;

  WriteBS(BS,I.MTypeTab[CMac^.MType]);              {* Write Mtype VLC *}
                    {* If MC then Save motion Vector Information. *}
  if CMac^.MType in [5..10] then
    begin
{if INFO.DEBUGVAL = 1 then
      write(' MC');}
      if MAC = 1 then
        begin
          MVDH := 0 + CMac^.MVH;
          MVDV := 0 + CMac^.MVV;
        end
      else
        begin
          if (MAC in [12,23]) or (not(PMac^.MType in [5..10])) then
            begin
              MVDH := CMac^.MVH;
              MVDV := CMac^.MVV;
            end
          else
            begin
              MVDH := CMac^.MVH - PMac^.MVH;
              MVDV := CMac^.MVV - PMac^.MVV;
            end;
        end;

      if MVDH < 0 then
```

```
        Idx1 := 32 + MVDH
      else
        Idx1 := MVDH;
      WriteBS(BS,I.MVDTab[Idx1]);


      if MVDV < 0 then
        Idx1 := 32 + MVDV
      else
        Idx1 := MVDV;
      WriteBS(BS,I.MVDTab[Idx1]);

    end;


  if CMac^.MType in [3,4,6,7,9,10] then
    WriteBS(BS,I.CBPTab[CMac^.CBP]);       {* Save CBP (transmit Blocks) *}

  if CMac^.MType in [1..4,6,7,9,10] then
    begin
      SaveCofs(I,BS,CMac);                 {* Save Coef.s In MAC *}
    end
  else
    begin
            {* In Case OF MC with zero Displacement then Do Nothing *}
    end;

end;



Procedure DCT_Quantise_Store(var I : Infotype; var UPic : Pictype; Fr_Num : word;var BS : Bstype);
{* This Procedure calls DCT TransForm, Quantise and Store Macoblocks for    *}
{* each macroblock.                                                        *}

var
  GOB,MAC : byte;
  Idx1,Idx2 : word;
  CMac,PMac : Macblkptrtype;

begin
  WritePicHeader(I,BS,Fr_Num);
  for GOB := 1 to Fr_Chars(I.Mov_Type,'G') do
    begin
      if (GOB > 1) and (Fr_Num = 1) then
        INFO.DEBUG := True
      else
        INFO.DEBUG := False;


      WriteGOBHeader(I,BS,GOB,UPic[GOB]^.GQuant);
      For MAC := 1 to MAC_GOB do
        begin
          CMac := @(UPic[GOB]^.Macs[MAC]);
          if MAC <> 1 then
            begin
              PMac := @(UPic[GOB]^.Macs[MAC - 1]);
              WriteMAC(I,BS,MAC,CMac,PMac,UPic[GOB]^.GQuant);
```

```
            end
          else
            begin
              WriteMAC(I,BS,MAC,CMac,PMac,UPic[GOB]^.GQuant);
            end;

        end;
    end;
end;


function DeQuantise (Value : Picvaltype; Quant : byte) : DCTvaltype;
{* Performs De quantisation According to CCITT standard.          *}

var
  Tmp : DCTvaltype;

begin
  if Value = 0 then
    DeQuantise := 0
  else
    begin
      if (Quant mod 2) = 1 then
        begin
          if Value > 0 then
            Tmp := (2*Value+1)*Quant
          else
            Tmp := (2*Value-1)*Quant;
        end
      else
        begin
          if Value > 0 then
            Tmp := (2*Value+1)*Quant - 1
          else
            Tmp := (2*Value-1)*Quant + 1;
        end;

      DeQuantise := Tmp;
    end;
end;

function Clip2(Value : DCTvaltype) : Rawpicvaltype;
{* Performs CCITT Clipping On For Raw Pictur Format after DCT and MC.    *}

var
  Tmp : integer;

begin
      Tmp := round(Value);

      if Tmp > 255 then
        Clip2 := 255
      else
        if Tmp < 0 then
          Clip2 := 0
        else
          Clip2 := Tmp;
end;
```

Appendix A

```
function Clip(Value : DCTvaltype) : Picvaltype;
{* Performs CCITT Clipping for Values for work frame after DCT and Quantise *}

begin
      Clip := round(Value);

      if Value > 255 then
        Clip := 255;

      if Value < -256 then
        Clip := -256;
end;

procedure DEQ_IDCT(var UMac : Macblkptrtype; Quant : byte; GQuant : boolean);
{* This Procedure Does The IDCT and calls De Quantisation.              *}

var
  Mult,Mult2,Calc : DCTvaltype;
  vSum1,uSum1 : DCTvaltype;
  vSum2,uSum2 : DCTvaltype;
  vSum3,uSum3 : DCTvaltype;
  vSum4,uSum4 : DCTvaltype;
  vSum5,uSum5 : DCTvaltype;
  vSum6,uSum6 : DCTvaltype;
  u,v,i,j,DSiz : byte;
  NMac : MacBlkptrtype;

begin
  Mult := 1/sqrt(2);
  Mult2 := 1/4;
  DSiz := BLK_WID + BLK_HIG;
  new (NMac);

  with UMac^ do
  for i := 0 to (BLK_HIG - 1) do
    begin
      for j := 0 to (BLK_WID - 1) do
        begin
          uSum1 := 0;
          uSum2 := 0;
          uSum3 := 0;
          uSum4 := 0;
          uSum5 := 0;
          uSum6 := 0;
          for u := 0 to (BLK_HIG - 1) do
            begin
              vSum1 := 0;
              vSum2 := 0;
              vSum3 := 0;
              vSum4 := 0;
              vSum5 := 0;
              vSum6 := 0;
              for v := 0 to (BLK_WID - 1) do
                begin
                  Calc := cos((2*i + 1)*u*pi/DSiz) *
                       cos((2*j + 1)*v*pi/DSiz);
```

```
                    if v = 0 then
                      Calc := Calc * Mult;

                    if u = 0 then
                      Calc := Calc * Mult;

                    vSum1 := vSum1 + (DeQuantise(RawMacYArray[(u+1),(v+1)],Quant) * Calc);
                    vSum2 := vSum2 + (DeQuantise(RawMacYArray[(u+1),(v+1+BLK_WID)],Quant) *
Calc);
                    vSum3 := vSum3 + (DeQuantise(RawMacYArray[(u+1+BLK_HIG),(v+1)],Quant) *
Calc);
                    vSum4 := vSum4 +
(DeQuantise(RawMacYArray[(u+1+BLK_HIG),(v+1+BLK_WID)],Quant) * Calc);

                    vSum5 := vSum5 + (DeQuantise(RawMacUArray[(u+1),(v+1)],Quant) * Calc);
                    vSum6 := vSum6 + (DeQuantise(RawMacVArray[(u+1),(v+1)],Quant) * Calc);
                  end;

                uSum1 := uSum1 + vSum1;
                uSum2 := uSum2 + vSum2;
                uSum3 := uSum3 + vSum3;
                uSum4 := uSum4 + vSum4;
                uSum5 := uSum5 + vSum5;
                uSum6 := uSum6 + vSum6;

              end;


            uSum1 := uSum1 * Mult2;
            uSum2 := uSum2 * Mult2;
            uSum3 := uSum3 * Mult2;
            uSum4 := uSum4 * Mult2;
            uSum5 := uSum5 * Mult2;
            uSum6 := uSum6 * Mult2;



            NMac^.RawMacYArray[i+1,j+1] := Clip(uSum1);
            NMac^.RawMacYArray[i+1,j+1+BLK_WID] := Clip(uSum2);
            NMac^.RawMacYArray[i+1+BLK_HIG,j+1] := Clip(uSum3);
            NMac^.RawMacYArray[i+1+BLK_HIG,j+1+BLK_WID] := Clip(uSum4);
            NMac^.RawMacUArray[i+1,j+1] := Clip(uSum5);
            NMac^.RawMacVArray[i+1,j+1] := Clip(uSum6);
          end;
      end;

    UMac^.RawMacYArray := NMac^.RawMacYArray;
    UMac^.RawMacUArray := NMac^.RawMacUArray;
    UMac^.RawMacVArray := NMac^.RawMacVArray;
    dispose(NMac);
end;


procedure DeQuant_IDCT(var I : Infotype; var UPic : Pictype);
{* This procedure Calls IDCT and De Quantise for every macro block.          *}
```

```
var
  GOB,MAC : byte;
  Idx1,Idx2 : word;
  CMac : Macblkptrtype;
  MCA : Mcatype;
  Top,Bottom,Left,Right : boolean;

begin
  for GOB := 1 to Fr_Chars(I.Mov_Type,'G') do
    begin
      For MAC := 1 to MAC_GOB do
        begin
          write(':');
          CMac := @(UPic[GOB]^.Macs[MAC]);
          DEQ_IDCT(CMac,UPic[GOB]^.GQuant,True);
        end;
    end;
end;

function DeMCSearch(var MCA : Mcatype; var UMac : Macblkptrtype; var CMac : RMacptrtype) :
boolean;
{* This Function Undoes the search done by MC Search to give the DCT and    *}
{* Quantisation Altered value to use as the previous picture.              *}

var
  Pos : word;
  MAC,Row,Col : byte;
  Idx1,Idx2 : byte;

begin
  DeMCSearch := False;

  if UMac^.MType in [1..2] then
    begin
      write('~');
      DeMCSearch := True;
      for Row := 1 to (BLK_HIG*2) do
        for Col := 1 to (BLK_WID*2) do
          begin
            CMac^.RawMacYArray[Row,Col] := Clip2(UMac^.RawMacYArray[Row,Col]);
            if ((Row mod 2) = 0) and ((Col mod 2) = 0) then
              begin
                CMac^.RawMacUArray[Row div 2,Col div 2] :=
                Clip2(UMac^.RawMacUArray[Row div 2,Col div 2]);
                CMac^.RawMacVArray[Row div 2,Col div 2] :=
                Clip2(UMac^.RawMacVArray[Row div 2,Col div 2]);
              end;
          end;
    end;

  if UMac^.MType in [3..4] then
    begin
      write('!');
      for Row := 1 to (BLK_HIG) do
        for Col := 1 to (BLK_WID) do
          begin
            if (UMac^.CBP and 32) = 32 then
              begin
```

```
            CMac^.RawMacYArray[Row,Col] :=
            Clip2(UMac^.RawMacYArray[Row,Col] + MCA[5]^.RawMacYArray[Row,Col]/1);
          end
        else
          begin
            CMac^.RawMacYArray[Row,Col] := Clip2(MCA[5]^.RawMacYArray[Row,Col]/1);
          end;

        if (UMac^.CBP and 16) = 16 then
          begin
            CMac^.RawMacYArray[Row,Col+BLK_WID] :=
            Clip2(UMac^.RawMacYArray[Row,Col+BLK_WID]
            + MCA[5]^.RawMacYArray[Row,Col+BLK_WID]/1);
          end
        else
          begin
            CMac^.RawMacYArray[Row,Col+BLK_WID] :=
Clip2(MCA[5]^.RawMacYArray[Row,Col+BLK_WID]/1);
          end;

        if (UMac^.CBP and 8) = 8 then
          begin
            CMac^.RawMacYArray[Row+BLK_HIG,Col] :=
            Clip2(UMac^.RawMacYArray[Row+BLK_HIG,Col]
            + MCA[5]^.RawMacYArray[Row+BLK_HIG,Col]/1);
          end
        else
          begin
            CMac^.RawMacYArray[Row+BLK_HIG,Col] :=
Clip2(MCA[5]^.RawMacYArray[Row+BLK_HIG,Col]/1);
          end;

        if (UMac^.CBP and 4) = 4 then
          begin
            CMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID] :=
            Clip2(UMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]
            + MCA[5]^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]/1);
          end
        else
          begin
            CMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID] :=
            Clip2(MCA[5]^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]/1);
          end;

        if (UMac^.CBP and 2) = 2 then
          begin
            CMac^.RawMacUArray[Row,Col] :=
            Clip2(UMac^.RawMacUArray[Row,Col]
            + MCA[5]^.RawMacUArray[Row,Col]/1);
          end
        else
          begin
            CMac^.RawMacUArray[Row,Col] :=
            Clip2(MCA[5]^.RawMacUArray[Row,Col]/1);
          end;

        if (UMac^.CBP and 1) = 1 then
          begin
```

```
            CMac^.RawMacVArray[Row,Col] :=
            Clip2(UMac^.RawMacVArray[Row,Col]
            + MCA[5]^.RawMacVArray[Row,Col]/1);
          end
        else
          begin
            CMac^.RawMacVArray[Row,Col] :=
            Clip2(MCA[5]^.RawMacVArray[Row,Col]/1);
          end;
      end;


  end;

if UMac^.MType in [5,8] then
  begin
    write('%');
    for Row := 1 to (BLK_HIG*2) do
      for Col := 1 to (BLK_WID*2) do
        begin
          Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
            + (BLK_WID * 2 * (Row-1) * 3) + (Col-1);

          ConMCRData(Pos,MAC,Idx1,Idx2,True);

          CMac^.RawMacYArray[Row,Col] := Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]/1);
        end;

    for Row := 1 to (BLK_HIG) do
      for Col := 1 to (BLK_WID) do
        begin
          Pos := ZRO_POS1 + (UMac^.MVH div 2) + (3 * BLK_WID * (UMac^.MVV div 2))
            + (BLK_WID * (Row-1) * 3) + (Col-1);

          ConMCRData(Pos,MAC,Idx1,Idx2,False);

          CMac^.RawMacUArray[Row,Col] := Clip2(MCA[MAC]^.RawMacUArray[Idx1,Idx2]/1);
          CMac^.RawMacVArray[Row,Col] := Clip2(MCA[MAC]^.RawMacVArray[Idx1,Idx2]/1);
        end;
  end;

if UMac^.MType in [6,7,9,10] then
  begin
    write('+');
    for Row := 1 to (BLK_HIG) do
      for Col := 1 to (BLK_WID) do
        begin

          Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
              + (BLK_WID * 2 * (Row-1) * 3) + (Col-1);
          ConMCRData(Pos,MAC,Idx1,Idx2,True);

          if (UMac^.CBP and 32) = 32 then
            begin
              CMac^.RawMacYArray[Row,Col] := Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]
              + UMac^.RawMacYArray[Row,Col]/1);
            end
          else
```

```
            begin
              CMac^.RawMacYArray[Row,Col] := Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]);
            end;

          Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
              + (BLK_WID * 2 * (Row-1) * 3) + (Col-1+BLK_WID);
          ConMCRData(Pos,MAC,Idx1,Idx2,True);

          if (UMac^.CBP and 16) = 16 then
            begin
              CMac^.RawMacYArray[Row,Col+BLK_WID] :=
Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]
              + UMac^.RawMacYArray[Row,Col+BLK_WID]/1);
            end
          else
            begin
              CMac^.RawMacYArray[Row,Col+BLK_WID] :=
Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]);
            end;

          Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
              + (BLK_WID * 2 * (Row-1+BLK_HIG) * 3) + (Col-1);
          ConMCRData(Pos,MAC,Idx1,Idx2,True);

          if (UMac^.CBP and 8) = 8 then
            begin
              CMac^.RawMacYArray[Row+BLK_HIG,Col] :=
Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]
              + UMac^.RawMacYArray[Row+BLK_HIG,Col]/1);
            end
          else
            begin
              CMac^.RawMacYArray[Row+BLK_HIG,Col] :=
Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]);
            end;

          Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
              + (BLK_WID * 2 * (Row-1+BLK_HIG) * 3) + (Col-1+BLK_WID);
          ConMCRData(Pos,MAC,Idx1,Idx2,True);

          if (UMac^.CBP and 4) = 4 then
            begin
              CMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID] :=
              Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]
              + UMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]/1);
            end
          else
            begin
              CMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID] :=
              Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]);
            end;

          Pos := ZRO_POS1 + (UMac^.MVH div 2) + (3 * BLK_WID * (UMac^.MVV div 2))
              + (BLK_WID * (Row-1) * 3) + (Col-1);

          ConMCRData(Pos,MAC,Idx1,Idx2,False);

          if (UMac^.CBP and 2) = 2 then
```

```
            begin
              CMac^.RawMacUArray[Row,Col] := Clip2(MCA[MAC]^.RawMacUArray[Idx1,Idx2]
                + UMac^.RawMacUArray[Row,Col]/1);
            end
          else
            begin
              CMac^.RawMacUArray[Row,Col] := Clip2(MCA[MAC]^.RawMacUArray[Idx1,Idx2]);
            end;
          if (UMac^.CBP and 1) = 1 then
            begin
              CMac^.RawMacVArray[Row,Col] := Clip2(MCA[MAC]^.RawMacVArray[Idx1,Idx2]
                + UMac^.RawMacVArray[Row,Col]/1);
            end
          else
            begin
              CMac^.RawMacVArray[Row,Col] := Clip2(MCA[MAC]^.RawMacVArray[Idx1,Idx2]);
            end;
        end;
    end;
end;


procedure DeMotion (var I : Infotype; var UPic : Pictype;var CPic,PPic : Rpictype; Pfe : boolean);
{* This Procedure Provides the surrounding blocks for the previous picture *}
{* to Undo the Motion Search.                                              *}

var
  Idx1,Idx2 : word;
  GOB,MAC,GOB1,MAC1,MAC_Num : word;
  MCA : Mcatype;
  Top,Bottom,Left,Right : boolean;
  UMac : Macblkptrtype;
  Intra : boolean;
  CMac : Rmacptrtype;


begin
  if not (I.Mov_Type in [CIF,QCIF,NTSC]) then
    Die ('Illigal Picture type (MotionAnalysis)',3);

  MAC_Num := Fr_Chars(I.Mov_Type,'M');
  for Idx1 := 1 to MAC_Num do
    begin
      ConvToM_G(I.Mov_Type,Idx1,GOB,MAC);
      UMac := @((UPic[GOB])^.Macs[MAC]);
      CMac := @((CPic[GOB])^.Macs[MAC]);

      if I.Mov_Type = CIF then
        begin
          If (GOB mod 2) = 1 then
            begin
              if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
                begin
                  Left := False;
                end
              else
                begin
```

```
        Left := True;
      end;
  end
else
  begin
    if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
          begin
            Right := False;
          end
        else
          begin
            Right := True;
          end;
      end;

    If GOB in [1,2] then
      begin
        if MAC in [1..GOB_WID] then
          begin
            Top := False;
          end
        else
          begin
            Top := True;
          end;
      end;

    If GOB in [11,12] then
      begin
        if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
          begin
            Bottom := False;
          end
        else
          begin
            Bottom := True;
          end;
      end;
    if not (GOB in [1,2,11,12]) then
      begin
        Top := True;
        Bottom := True;
      end;
  end;

if I.Mov_Type = QCIF then
  begin
    if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
      begin
        Left := False;
      end
    else
      begin
        Left := True;
      end;

    if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
      begin
```

```
        Right := False;
      end
    else
      begin
        Right := True;
      end;

    If GOB = 1 then
      begin
        if MAC in [1..GOB_WID] then
          begin
            Top := False;
          end
        else
          begin
            Top := True;
          end;
      end;

    If GOB = 3 then
      begin
        if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
          begin
            Bottom := False;
          end
        else
          begin
            Bottom := True;
          end;
      end;

    if not (GOB in [1,3]) then
      begin
        Top := True;
        Bottom := True;
      end;
  end;

if I.Mov_Type = NTSC then
  begin
    Left := True;
    Right := True;
    Top := True;
    Bottom := True;
    If (GOB mod 2) = 1 then
      begin
        if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
          begin
            Left := False;
          end
        else
          begin
            Left := True;
          end;
      end
    else
      begin
        if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
```

```
            begin
              Right := False;
            end
          else
            begin
              Right := True;
            end;
        end;

      If GOB in [1,2] then
        begin
          if MAC in [1..GOB_WID] then
            begin
              Top := False;
            end
          else
            begin
              Top := True;
            end;
        end;

      If GOB in [9,10] then
        begin
          if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
            begin
              Bottom := False;
            end
          else
            begin
              Bottom := True;
            end;
        end;

      if not (GOB in [1,2,9,10]) then
        begin
          Top := True;
          Bottom := True;
        end;
    end;

  if Top and (Pfe) then
    begin
      ConvToM_G(I.Mov_Type,(Idx1 - GOB_WID*2),GOB1,MAC1);
      MCA[2] := @((PPic[GOB1])^.Macs[MAC1]);
    end
  else
    MCA[2] := nil;

  if Top and Right and (Pfe) then
    begin
      ConvToM_G(I.Mov_Type,(Idx1 - GOB_WID*2 + 1),GOB1,MAC1);
      MCA[3] := @((PPic[GOB1])^.Macs[MAC1]);
    end
  else
    MCA[3] := nil;

  if Right and (Pfe) then
    begin
```

```
            ConvToM_G(I.Mov_Type,(Idx1 + 1),GOB1,MAC1);
            MCA[6] := @((PPic[GOB1])^.Macs[MAC1]);
          end
        else
          MCA[6] := nil;

        if Right and Bottom and (Pfe) then
          begin
            ConvToM_G(I.Mov_Type,(Idx1 + 1 + GOB_WID*2),GOB1,MAC1);
            MCA[9] := @((PPic[GOB1])^.Macs[MAC1]);
          end
        else
          MCA[9] := nil;

        if Bottom and (Pfe) then
          begin
            ConvToM_G(I.Mov_Type,(Idx1 + GOB_WID*2),GOB1,MAC1);
            MCA[8] := @((PPic[GOB1])^.Macs[MAC1]);
          end
        else
          MCA[8] := nil;

        if Bottom and Left and (Pfe) then
          begin
            ConvToM_G(I.Mov_Type,(Idx1 + GOB_WID*2 - 1),GOB1,MAC1);
            MCA[7] := @((PPic[GOB1])^.Macs[MAC1]);
          end
        else
          MCA[7] := nil;

        if Left and (Pfe) then
          begin
            ConvToM_G(I.Mov_Type,(Idx1 - 1),GOB1,MAC1);
            MCA[4] := @((PPic[GOB1])^.Macs[MAC1]);
          end
        else
          MCA[4] := nil;

        if Left and Top and (Pfe) then
          begin
            ConvToM_G(I.Mov_Type,(Idx1 - 1 - GOB_WID*2),GOB1,MAC1);
            MCA[1] := @((PPic[GOB1])^.Macs[MAC1]);
          end
        else
          MCA[1] := nil;

        if Pfe then
          MCA[5] := @((PPic[GOB])^.Macs[MAC]);

        Intra := DeMCSearch(MCA,UMac,CMac);

    end

end;

procedure BSEnd(var I : infotype; var BS : Bstype);
{* This Procedure stuffs the bitstream to make the file size rounded to a  *}
{* whole number of bytes (Flush the Bitstram) *}
```

```
var
  X,Count : byte;

begin
  case BS.VBits of
    0 : X := 0;
    1 : X := 5;
    2 : X := 2;
    3 : X := 7;
    4 : X := 4;
    5 : X := 1;
    6 : X := 6;
    7 : X := 3;
  end;

  if BS.VBits = 8 then
    WriteBS(BS,'1')

  else
    begin
      for Count := 1 to X do
        WriteBS(BS,I.MBATab[34]);                    {* MBA Stuffing *}
      WriteBS(BS,'1');
    end;

end;


procedure Encode(var I : Infotype);
{* This Procedure Does The Supa, Mega, Giga Encode Sequence.              *}

var
  Idx1 : word;
  UFrame : Pictype;
  CFrame,PFrame : Rpictype;
  PFe : boolean;
  BS :Bstype;

begin
  PFe := False;                    {* Initialise Previous Frame *}

  assign (BS.BsFile,I.C_File_Name);
  rewrite (BS.BsFile);

  BS.Vbits := 0;
  BS.CPos := 0;
  BS.Tr_Bits := '';

  for Idx1 := I.Start_Fr to I.End_Fr do
    begin
    {* ------------------------------------------- *}
      write('Loading Frame : ',Idx1);
      LoadFrame(I,CFrame,Idx1);
      writeln(' ... Complete.');

      write('Allocating Work Frame : ',Idx1);
      AllocFrame(UFrame,I.Mov_Type);
```

```
      writeln(' ... Complete.');
    {* ---------------------------------------- *}
      write('Motion Analysis on Frame : ',Idx1,' ');
      MotionAnalysis(I,UFrame,CFrame,PFrame,Pfe);
      writeln(' ... Complete.');
    {* ---------------------------------------- *}
      write('DCT Quantising And Storing Frame : ',Idx1,' ');
      DCT_Quantise_Store(I,UFrame,Idx1,BS);
      writeln(' Complete.');
    {* ---------------------------------------- *}
      if Idx1 < I.End_Fr then
        begin
          write('IDCT and De Quantising Frame : ',Idx1,' ');
          DeQuant_IDCT(I,UFrame);
          writeln(' Complete.');
    {* ---------------------------------------- *}
          write('De - Motion Analysis on Frame : ',Idx1,' ');
          DeMotion(I,UFrame,CFrame,F Frame,Pfe);
          writeln(' Complete.');
        end;
    {* ---------------------------------------- *}
      write('Un-Allocating Work Frame : ',Idx1);
      UnloadFrame(I,UFrame);
      writeln(' ... Complete.');
    {* ---------------------------------------- *}
      if Not PFe then
        begin
          PFrame := CFrame;
          PFe := True;
        end
      else
        begin
          RUnloadFrame(I,PFrame);
          PFrame := CFrame;
        end;
    {* ---------------------------------------- *}
    end;
  BSEnd(I,BS);
  close(BS.BsFile);
end;


begin {* Main Encode Program *}
  Init(Info);                    {* Init All Structures and Tables *}
  Encode(Info);                    {* Start the Encode Process      *}
  write('Program Terminated Gracefully, Press Enter To Exit ->');
  repeat
    write('');
  until keypressed;
  readln;
end.  {* Main Encode Program *}
```

Appendix B

```
program H261_Decode (input,output);
{*************************************************************************}
{* Program Name : H261_Decode                                            *}
{* Author : Geoffrey Alagoda, Student No : 0911697                       *}
{* Date of Last Modification : 14/October/1995                           *}
{* Purpose : To implement an Decoder based on the CCITT H.261 Motion Video *}
{*       Compression / Decompresson Recomendation. This is done for a    *}
{*       fourth year Engineering Project so that it can be later         *}
{*       improved upon. The implimentation is to be on an IBM PC         *}
{*       Compatible machine using Tutbo Pascal version 7 with at least   *}
{*       8 Mb of RAM and > 386 DX40. The final Encoded file produced     *}
{*       may differ slightly from other implementation as its based on   *}
{*       my interpretation of the H.261 Standard.                        *}
{*                                                                       *}
{* Program Modules: Motion Compensation / Estimation                     *}
{*           DCT / IDCT                                                  *}
{*           Quantisation / De-Quantisation                              *}
{*           Run Length Encoding                                         *}
{*           Variable Length Encoding                                    *}
{*           File Management Code                                        *}
{*           Memory Management Code - etc                                *}
{*************************************************************************}

uses crt,dos;          {* Standard Turbo Pascal Screen and File Library *}

const
   BLK_WID = 8;          {* The width of a block to be dealt with. *}
   BLK_HIG = 8;          {* The height of a block to be dealt with. *}

   GOB_WID = 11;                    {* MAC Blocks Per GOB Row *}
   GOB_HIG = 3;                     {* MAC Blocks Per GOB Col *}
   MAC_GOB = GOB_WID * GOB_HIG;     {* Number of Macroblocks per GOB *}

   MAX_GOBS = 12;                   {* Maximum Number Of GOBS *}
   MAX_CBP = 6;                     {* Max Blocks per Macroblock *}
   SRCH_WID = 8;                    {* MC Pel Search Size *}
   ZRO_POS = 3 * BLK_WID * 4 * BLK_HIG + BLK_WID * 2 + 1;
   ZRO_POS1 = 3 * BLK_WID * BLK_HIG + BLK_WID + 1;

   {* Picture Size Definitions *}

   P_CIF_YWID = 352;
   P_CIF_YHIG = 288;
   P_CIF_CWID = 176;
   P_CIF_CHIG = 144;
   CIF_GOB = 12;

   P_QCIF_YWID = 176;
   P_QCIF_YHIG = 144;
   P_QCIF_CWID = 88;
   P_QCIF_CHIG = 72;
   QCIF_GOB = 3;

   N_CIF_YWID = 352;
   N_CIF_YHIG = 240;
   N_CIF_CWID = 176;
   N_CIF_CHIG = 120;
   NTSC_GOB = 10;
```

Appendix B

```
  {* Picture Size Definitions *}

  {* Quantisation *}

  IGQUANT = 8;
  IMQUANT = 8;
  DCQUANT = IGQUANT;

  {* Picture Types *}

  CIF = 1;                                {* 12 GOBS *}
  QCIF = 2;                               {* 3 GOBS *}
  NTSC = 3;                               {* 10 GOBS *}

  {* Picture Types *}

type
  Rawpicvaltype = byte;        {* For Raw Picture Loading - saves memory *}
  Picvaltype = integer;        {* The Value Type For A Picture Level *}
  DCTvaltype = single;         {* The Value Type For A DCT Level *}


  Picblktype = array[1..BLK_WID,1..BLK_HIG] of Picvaltype;
  Rpicblktype = array[1..BLK_WID,1..BLK_HIG] of Rawpicvaltype;
                                {* Block Type Dec. *}
  Spicblktype = array[1..BLK_WID*2,1..BLK_HIG*2] of Picvaltype;
  Rspicblktype = array[1..BLK_WID*2,1..BLK_HIG*2] of Rawpicvaltype;
                                {* Super Block Type Dec. *}
  Blklinetype = array[1..BLK_WID*BLK_HIG] of Picvaltype;
                                {* Line After ZigZag Type Dec. *}
  Ftype = file of byte;                 {* Stream File Type *}

  Rmacptrtype = ^Rmactype;          {* Used to define Blocks with Bytes *}
  Rmactype = record
        RawMacYArray : Rspicblktype;
        RawMacUArray : Rpicblktype;
        RawMacVArray : Rpicblktype;
  end;
  Macblkptrtype = ^Macblktype;
                {* Used For Differential Blocks with integers *}
  Macblktype = record
         MType : byte;        {* Macro Blk type as in Table 2/H.261 *}
         MQuant : byte;      {* Quantisation if Mtype in 2,4,7 or 10 *}
         MVV,MVH : shortint;                {* Motion vector *}
         CBP : shortint;                {* Which To transmit *}
                              {* Blocks For Pic Info *}
         RawMacYArray : Spicblktype;
         RawMacUArray : Picblktype;
         RawMacVArray : Picblktype;
  end;

  Gobptrtype = ^Gobtype;
  Gobtype = record                      {* Group Of Blocks Type *}
       GQuant : byte;                   {* GOB Quantisation *}
       Macs : Array [1..MAC_GOB] of Macblktype;       {* Macro Blocks *}

  end;
```

Appendix B

```
Rgobptrtype = ^RGobtype;
RGobtype = record                          {* Group Of Blocks Type *}
      Macs : Array [1..MAC_GOB] of RMactype;        {* Macro Blocks *}

end;


Pictype = array[1..MAX_GOBS] of Gobptrtype;    {* An Integer work Frame *}
Rpictype = array[1..MAX_GOBS] of Rgobptrtype;      {* A raw byte frame *}
Mcatype = array [1..9] of Rmacptrtype; {* For MC The Surrounding Blocks *}

Bstype = record                                {* Bit Stream File *}
     VBits : byte;                       {* The Number Of Valid Bits *}
     CBits : byte;
     ChkVal : string;                        {* Not used In Encoding *}
     Tr_Bits : string;                   {* The Actual Transfere Bits *}
                    {* Not Used In Encoding Only For Decoding *}
     FEOF : boolean;
     BsFile : Ftype;
     FSize : single;
     CPos : single;

end;

Vlccoftype = record        {* Variable length Encoding For Coefficients *}
     COFF : string;
     Run : byte;
     Level : byte;
end;

Zztype = Array[1..63] of byte;
                  {* Zig Zag Lookup Table - Easier and Quicker *}

Infotype = record      {* Generally Contains Most Info About Everything *}
      L_File_Name,C_File_Name : string;
                     {* The Loose and Compressed File names *}
      Start_Fr,End_Fr : word;    {* Start Frame and End Frame Numbers *}
      Mov_Type : byte;             {* 2 = QCIF, 1 = CIF & 3 = NTSC *}
      P_Mov_Type : byte;               {* Previous Frame Mov_Type *}

      Split_Screen : Boolean;           {* Split Screen Indicator *}
      Doc_Cam_Ind : Boolean;                 {* Camera Indicator *}
      Fr_Pic_Rel : Boolean;            {* Reletive Pic Indicator *}
      TRef : Byte;                     {* Temporal Reference*}

      MBATab : Array[1..34] of string;          {* MBA Lookup Table *}
      MTypeTab : Array[1..10] of string;       {* MType Lookup Table *}
      MVDTab : Array[0..31] of string;     {* Motion Vec Lookup Table *}
      CBPTab : Array[1..63] of string;          {* Transfer Sequence *}
      COTab : Array[1..63] of Vlccoftype;       {* Coef. Lookup table *}
      ZZCol : Zztype;                  {* ZiZag Col. Lookup Table *}
      ZZRow : Zztype;                   {* ZiZag Row. Lookup Table *}
      Debug1 : boolean;
      DebugVal : byte;
end;                                        {* Infotype Defn *}
```

Appendix B

```
var                                    {* Global Variables *}
  Info : Infotype;                          {* Main Info *}

function FFName (InStr : string; Value : word; Ext : Char) : string;
{* This function Constructs filename strings for Loose files with values at *}
{* the end of the name                                                      *}

var
  TmpStr : String;

begin
  str(Value:0,TmpStr);
  FFName := InStr+TmpStr+'.'+Ext;
end;

function FileExists (Nam : string) : Single;
{* This function looks to see if a file exists and returns the file size.  *}

var
  F : file of byte;
  At : word;
  F_Size : longint;

begin
  assign(F,Nam);
  GetFAttr(F, At);
  if DOSERROR <> 0 then
    FileExists := 0
  else
    begin
      reset(F);
      F_Size := filesize(F);
      FileExists := F_Size;
    end;
  if DOSERROR = 0 then
    close(F);
end;

procedure Die(Msg : string ; Code : byte);
{* This procedure is to give an error message when an error occurs and     *}
{* un-gracefully Quit The program with the error code.                     *}

begin
  writeln;
  writeln;
  if Msg = '' then
    writeln('Error - Dowh, this is new one, dunno what this is')
  else
    writeln('Error ',Code,': ',Msg);
  write('Program Crashed, Please Hit Enter ->');
  readln;
  halt(Code);
end;

function BtS(InVal : Byte) : string;
{* This Function converts InVal in to a string of bits.                   *}

var
```

```
  str1 : string[8];


begin
  str1 := '';
  if (InVal and 128) = 128 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 64) = 64 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 32) = 32 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 16) = 16 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 8) = 8 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 4) = 4 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 2) = 2 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';
  if (InVal and 1) = 1 then
    str1 := str1 + '1'
  else
    str1 := str1 + '0';

  BtS := str1;
end;


function StB(InVal : String) : byte;
{* This Function Converts the InVal String of bits to a byte.          *}

var
  Out : byte;
  Idx1 : byte;
  Len : Byte;
  TmpStr : string;

begin
  Out := 0;
  Len := Length(InVal);
  TmpStr := '';

  for Idx1 := 1 to (8 - Len) do
    TmpStr := TmpStr + '0';
```

```
  TmpStr := TmpStr + InVal;

  for Idx1 := 1 to 8 do
  if TmpStr[Idx1] = '1' then
    case Idx1 of
      1 : Out := Out + 128;
      2 : Out := Out + 64;
      3 : Out := Out + 32;
      4 : Out := Out + 16;
      5 : Out := Out + 8;
      6 : Out := Out + 4;
      7 : Out := Out + 2;
      8 : Out := Out + 1;
    end;
  StB := Out;
end;

{* ---------------------------------------------------------------- *}

procedure Init(var I : Infotype);
{* This Procedure Initialises the Main Info with Data such as VLC Table    *}
{* Codes, Filenames.. etc.                                                 *}

var
  InStr : string;
  Result : integer;
  Idx : word;
  FExists : boolean;
  FSiz : single;

begin
  clrscr;                {* Turbo Pascal's Famous Clear Screen Routine *}
  writeln('H.261 Motion Video Compression Decoder V 0.99 (Geoffrey Alagoda - 1995)');
  writeln;
  repeat
    write('Please Enter The Compresed File Name ->');
    InStr := 'C:\AA\bike.261';
    {readln(InStr);}
    writeln(InStr);
    I.C_File_Name := InStr;
    if FileExists(InStr) < 1  then
      begin
        writeln('<><><><> Sorry, File Not Found <><><><>');
        InStr := '';
      end;
  until (InStr <> '');


  repeat
    writeln('Please Enter Uncompressed File Name common to all Frames');
    write('->');
    {readln(InStr);}
    InStr := 'C:\AA\A';
    writeln(InStr);
    I.L_File_Name := InStr;
    til InStr <> '';
```

Appendix B

```
I.Debug1 := False;
I.DebugVal := 0;


writeln('Initialising Tables');

I.MBATab[1] := '1';
I.MBATab[2] := '011';
I.MBATab[3] := '010';
I.MBATab[4] := '0011';
I.MBATab[5] := '0010';
I.MBATab[6] := '00011';
I.MBATab[7] := '00010';
I.MBATab[8] := '0000111';
I.MBATab[9] := '0000110';
I.MBATab[10] := '00001011';
I.MBATab[11] := '00001010';
I.MBATab[12] := '00001001';
I.MBATab[13] := '00001000';
I.MBATab[14] := '00000111';
I.MBATab[15] := '00000110';
I.MBATab[16] := '0000010111';
I.MBATab[17] := '0000010110';
I.MBATab[18] := '0000010101';
I.MBATab[19] := '0000010100';
I.MBATab[20] := '0000010011';
I.MBATab[21] := '0000010010';
I.MBATab[22] := '00000100011';
I.MBATab[23] := '00000100010';
I.MBATab[24] := '00000100001';
I.MBATab[25] := '00000100000';
I.MBATab[26] := '00000011111';
I.MBATab[27] := '00000011110';
I.MBATab[28] := '00000011101';
I.MBATab[29] := '00000011100';
I.MBATab[30] := '00000011011';
I.MBATab[31] := '00000011010';
I.MBATab[32] := '00000011001';
I.MBATab[33] := '00000011000';
I.MBATab[34] := '00000001111';

I.MTypeTab[1] := '0001';
I.MTypeTab[2] := '0000001';
I.MTypeTab[3] := '1';
I.MTypeTab[4] := '00001';
I.MTypeTab[5] := '000000001';
I.MTypeTab[6] := '00000001';
I.MTypeTab[7] := '0000000001';
I.MTypeTab[8] := '001';
I.MTypeTab[9] := '01';
I.MTypeTab[10] := '000001';

I.MVDTab[0] := '1';
I.MVDTab[1] := '010';
I.MVDTab[2] := '0010';
I.MVDTab[3] := '00010';
I.MVDTab[4] := '0000110';
I.MVDTab[5] := '00001010';
```

Appendix B

```
I.MVDTab[6] := '00001000';
I.MVDTab[7] := '00000110';
I.MVDTab[8] := '0000010110';
I.MVDTab[9] := '0000010100';
I.MVDTab[10] := '0000010010';
I.MVDTab[11] := '00000100010';
I.MVDTab[12] := '00000100000';
I.MVDTab[13] := '00000011110';
I.MVDTab[14] := '00000011100';
I.MVDTab[15] := '00000011010';
I.MVDTab[16] := '00000011001';
I.MVDTab[17] := '00000011011';
I.MVDTab[18] := '00000011101';
I.MVDTab[19] := '00000011111';
I.MVDTab[20] := '00000100001';
I.MVDTab[21] := '00000100011';
I.MVDTab[22] := '0000010011';
I.MVDTab[23] := '0000010101';
I.MVDTab[24] := '0000010111';
I.MVDTab[25] := '00000111';
I.MVDTab[26] := '00001001';
I.MVDTab[27] := '00001011';
I.MVDTab[28] := '0000111';
I.MVDTab[29] := '00011';
I.MVDTab[30] := '0011';
I.MVDTab[31] := '011';

I.CBPTab[60] := '111';
I.CBPTab[4] := '1101';
I.CBPTab[8] := '1100';
I.CBPTab[16] := '1011';
I.CBPTab[32] := '1010';
I.CBPTab[12] := '10011';
I.CBPTab[48] := '10010';
I.CBPTab[20] := '10001';
I.CBPTab[40] := '10000';
I.CBPTab[28] := '01111';
I.CBPTab[44] := '01110';
I.CBPTab[52] := '01101';
I.CBPTab[56] := '01100';
I.CBPTab[1] := '01011' ;
I.CBPTab[61] := '01010';
I.CBPTab[2] := '01001' ;
I.CBPTab[62] := '01000';
I.CBPTab[24] := '001111';
I.CBPTab[36] := '001110';
I.CBPTab[3] := '001101' ;
I.CBPTab[63] := '001100';
I.CBPTab[5] := '0010111';
I.CBPTab[9] := '0010110';
I.CBPTab[17] := '0010101';
I.CBPTab[33] := '0010100';
I.CBPTab[6] := '0010011' ;
I.CBPTab[10] := '0010010';
I.CBPTab[18] := '0010001';
I.CBPTab[34] := '0010000';
I.CBPTab[7] := '00011111';
I.CBPTab[11] := '00011110';
```

```
I.CBPTab[19] := '00011101';
I.CBPTab[35] := '00011100';
I.CBPTab[13] := '00011011';
I.CBPTab[49] := '00011010';
I.CBPTab[21] := '00011001';
I.CBPTab[41] := '00011000';
I.CBPTab[14] := '00010111';
I.CBPTab[50] := '00010110';
I.CBPTab[22] := '00010101';
I.CBPTab[42] := '00010100';
I.CBPTab[15] := '00010011';
I.CBPTab[51] := '00010010';
I.CBPTab[23] := '00010001';
I.CBPTab[43] := '00010000';
I.CBPTab[25] := '00001111';
I.CBPTab[37] := '00001110';
I.CBPTab[26] := '00001101';
I.CBPTab[38] := '00001100';
I.CBPTab[29] := '00001011';
I.CBPTab[45] := '00001010';
I.CBPTab[53] := '00001001';
I.CBPTab[57] := '00001000';
I.CBPTab[30] := '00000111';
I.CBPTab[46] := '00000110';
I.CBPTab[54] := '00000101';
I.CBPTab[58] := '00000100';
I.CBPTab[31] := '000000111';
I.CBPTab[47] := '000000110';
I.CBPTab[55] := '000000101';
I.CBPTab[59] := '000000100';
I.CBPTab[27] := '000000011';
I.CBPTab[39] := '000000010';


(I.COTab[1]).COFF := '000001';
(I.COTab[1]).Run := 27;
(I.COTab[1]).Level := 1;

(I.COTab[2]).COFF := '011';
(I.COTab[2]).Run := 1;
(I.COTab[2]).Level := 1;

(I.COTab[3]).COFF := '0100';
(I.COTab[3]).Run := 0;
(I.COTab[3]).Level := 2;

(I.COTab[4]).COFF := '0101';
(I.COTab[4]).Run := 2;
(I.COTab[4]).Level := 1;

(I.COTab[5]).COFF := '00101';
(I.COTab[5]).Run := 0;
(I.COTab[5]).Level := 3;

(I.COTab[6]).COFF := '00111';
(I.COTab[6]).Run := 3;
(I.COTab[6]).Level := 1;
```

```
(I.COTab[7]).COFF := '00110';
(I.COTab[7]).Run := 4;
(I.COTab[7]).Level := 1;

(I.COTab[8]).COFF := '000110';
(I.COTab[8]).Run := 1;
(I.COTab[8]).Level := 2;

(I.COTab[9]).COFF := '000111';
(I.COTab[9]).Run := 5;
(I.COTab[9]).Level := 1;

(I.COTab[10]).COFF := '000101';
(I.COTab[10]).Run := 6;
(I.COTab[10]).Level := 1;

(I.COTab[11]).COFF := '000100';
(I.COTab[11]).Run := 7;
(I.COTab[11]).Level := 1;

(I.COTab[12]).COFF := '0000110';
(I.COTab[12]).Run := 0;
(I.COTab[12]).Level := 4;

(I.COTab[13]).COFF := '0000100';
(I.COTab[13]).Run := 2;
(I.COTab[13]).Level := 2;

(I.COTab[14]).COFF := '0000111';
(I.COTab[14]).Run := 8;
(I.COTab[14]).Level := 1;

(I.COTab[15]).COFF := '0000101';
(I.COTab[15]).Run := 9;
(I.COTab[15]).Level := 1;

(I.COTab[16]).COFF := '00100110';
(I.COTab[16]).Run := 0;
(I.COTab[16]).Level := 5;

(I.COTab[17]).COFF := '00100001';
(I.COTab[17]).Run := 0;
(I.COTab[17]).Level := 6;

(I.COTab[18]).COFF := '00100101';
(I.COTab[18]).Run := 1;
(I.COTab[18]).Level := 3;

(I.COTab[19]).COFF := '00100100';
(I.COTab[19]).Run := 3;
(I.COTab[19]).Level := 2;

(I.COTab[20]).COFF := '00100111';
(I.COTab[20]).Run := 10;
(I.COTab[20]).Level := 1;

(I.COTab[21]).COFF := '00100011';
(I.COTab[21]).Run := 11;
```

```
(I.COTab[21]).Level := 1;

(I.COTab[22]).COFF := '00100010';
(I.COTab[22]).Run := 12;
(I.COTab[22]).Level := 1;

(I.COTab[23]).COFF := '00100000';
(I.COTab[23]).Run := 13;
(I.COTab[23]).Level := 1;

(I.COTab[24]).COFF := '0000001010';
(I.COTab[24]).Run := 0;
(I.COTab[24]).Level := 7;

(I.COTab[25]).COFF := '0000001100';
(I.COTab[25]).Run := 1;
(I.COTab[25]).Level := 4;

(I.COTab[26]).COFF := '0000001011';
(I.COTab[26]).Run := 2;
(I.COTab[26]).Level := 3;

(I.COTab[27]).COFF := '0000001111';
(I.COTab[27]).Run := 4;
(I.COTab[27]).Level := 2;

(I.COTab[28]).COFF := '0000001001';
(I.COTab[28]).Run := 5;
(I.COTab[28]).Level := 2;

(I.COTab[29]).COFF := '0000001110';
(I.COTab[29]).Run := 14;
(I.COTab[29]).Level := 1;

(I.COTab[30]).COFF := '0000001101';
(I.COTab[30]).Run := 15;
(I.COTab[30]).Level := 1;

(I.COTab[31]).COFF := '0000001000';
(I.COTab[31]).Run := 16;
(I.COTab[31]).Level := 1;

(I.COTab[32]).COFF := '000000011101';
(I.COTab[32]).Run := 0;
(I.COTab[32]).Level := 8;

(I.COTab[33]).COFF := '000000011000';
(I.COTab[33]).Run := 0;
(I.COTab[33]).Level := 9;

(I.COTab[34]).COFF := '000000010011';
(I.COTab[34]).Run := 0;
(I.COTab[34]).Level := 10;

(I.COTab[35]).COFF := '000000010000';
(I.COTab[35]).Run := 0;
(I.COTab[35]).Level := 11;
```

Appendix B

```
(I.COTab[36]).COFF := '000000011011';
(I.COTab[36]).Run := 1;
(I.COTab[36]).Level := 5;

(I.COTab[37]).COFF := '000000010100';
(I.COTab[37]).Run := 2;
(I.COTab[37]).Level := 4;

(I.COTab[38]).COFF := '000000011100';
(I.COTab[38]).Run := 3;
(I.COTab[38]).Level := 3;

(I.COTab[39]).COFF := '000000010010';
(I.COTab[39]).Run := 4;
(I.COTab[39]).Level := 3;

(I.COTab[40]).COFF := '000000011110';
(I.COTab[40]).Run := 6;
(I.COTab[40]).Level := 2;

(I.COTab[41]).COFF := '000000010101';
(I.COTab[41]).Run := 7;
(I.COTab[41]).Level := 2;

(I.COTab[42]).COFF := '000000010001';
(I.COTab[42]).Run := 8;
(I.COTab[42]).Level := 2;

(I.COTab[43]).COFF := '000000011111';
(I.COTab[43]).Run := 17;
(I.COTab[43]).Level := 1;

(I.COTab[44]).COFF := '000000011010';
(I.COTab[44]).Run := 18;
(I.COTab[44]).Level := 1;

(I.COTab[45]).COFF := '000000011001';
(I.COTab[45]).Run := 19;
(I.COTab[45]).Level := 1;

(I.COTab[46]).COFF := '000000010111';
(I.COTab[46]).Run := 20;
(I.COTab[46]).Level := 1;

(I.COTab[47]).COFF := '000000010110';
(I.COTab[47]).Run := 21;
(I.COTab[47]).Level := 1;

(I.COTab[48]).COFF := '0000000011010';
(I.COTab[48]).Run := 0;
(I.COTab[48]).Level := 12;

(I.COTab[49]).COFF := '0000000011001';
(I.COTab[49]).Run := 0;
(I.COTab[49]).Level := 13;

(I.COTab[50]).COFF := '0000000011000';
(I.COTab[50]).Run := 0;
```

```
(I.COTab[50]).Level := 14;

(I.COTab[51]).COFF := '0000000010111';
(I.COTab[51]).Run := 0;
(I.COTab[51]).Level := 15;

(I.COTab[52]).COFF := '0000000010110';
(I.COTab[52]).Run := 1;
(I.COTab[52]).Level := 6;

(I.COTab[53]).COFF := '0000000010101';
(I.COTab[53]).Run := 1;
(I.COTab[53]).Level := 7;

(I.COTab[54]).COFF := '0000000010100';
(I.COTab[54]).Run := 2;
(I.COTab[54]).Level := 5;

(I.COTab[55]).COFF := '0000000010011';
(I.COTab[55]).Run := 3;
(I.COTab[55]).Level := 4;

(I.COTab[56]).COFF := '0000000010010';
(I.COTab[56]).Run := 5;
(I.COTab[56]).Level := 3;

(I.COTab[57]).COFF := '0000000010001';
(I.COTab[57]).Run := 9;
(I.COTab[57]).Level := 2;

(I.COTab[58]).COFF := '0000000010000';
(I.COTab[58]).Run := 10;
(I.COTab[58]).Level := 2;

(I.COTab[59]).COFF := '0000000011111';
(I.COTab[59]).Run := 22;
(I.COTab[59]).Level := 1;

(I.COTab[60]).COFF := '0000000011110';
(I.COTab[60]).Run := 23;
(I.COTab[60]).Level := 1;

(I.COTab[61]).COFF := '0000000011101';
(I.COTab[61]).Run := 24;
(I.COTab[61]).Level := 1;

(I.COTab[62]).COFF := '0000000011100';
(I.COTab[62]).Run := 25;
(I.COTab[62]).Level := 1;

(I.COTab[63]).COFF := '0000000011011';
(I.COTab[63]).Run := 26;
(I.COTab[63]).Level := 1;


I.ZZRow[1] := 0; I.ZZCol[1] := 1;
I.ZZRow[2] := 1; I.ZZCol[2] := 0;
I.ZZRow[3] := 2; I.ZZCol[3] := 0;
```

Appendix B

```
I.ZZRow[4] := 1; I.ZZCol[4] := 1;
I.ZZRow[5] := 0; I.ZZCol[5] := 2;
I.ZZRow[6] := 0; I.ZZCol[6] := 3;
I.ZZRow[7] := 1; I.ZZCol[7] := 2;
I.ZZRow[8] := 2; I.ZZCol[8] := 1;
I.ZZRow[9] := 3; I.ZZCol[9] := 0;
I.ZZRow[10] := 4; I.ZZCol[10] := 0;
I.ZZRow[11] := 3; I.ZZCol[11] := 1;
I.ZZRow[12] := 2; I.ZZCol[12] := 2;
I.ZZRow[13] := 1; I.ZZCol[13] := 3;
I.ZZRow[14] := 0; I.ZZCol[14] := 4;
I.ZZRow[15] := 0; I.ZZCol[15] := 5;
I.ZZRow[16] := 1; I.ZZCol[16] := 4;
I.ZZRow[17] := 2; I.ZZCol[17] := 3;
I.ZZRow[18] := 3; I.ZZCol[18] := 2;
I.ZZRow[19] := 4; I.ZZCol[19] := 1;
I.ZZRow[20] := 5; I.ZZCol[20] := 0;
I.ZZRow[21] := 6; I.ZZCol[21] := 0;
I.ZZRow[22] := 5; I.ZZCol[22] := 1;
I.ZZRow[23] := 4; I.ZZCol[23] := 2;
I.ZZRow[24] := 3; I.ZZCol[24] := 3;
I.ZZRow[25] := 2; I.ZZCol[25] := 4;
I.ZZRow[26] := 1; I.ZZCol[26] := 5;
I.ZZRow[27] := 0; I.ZZCol[27] := 6;
I.ZZRow[28] := 0; I.ZZCol[28] := 7;
I.ZZRow[29] := 1; I.ZZCol[29] := 6;
I.ZZRow[30] := 2; I.ZZCol[30] := 5;
I.ZZRow[31] := 3; I.ZZCol[31] := 4;
I.ZZRow[32] := 4; I.ZZCol[32] := 3;
I.ZZRow[33] := 5; I.ZZCol[33] := 2;
I.ZZRow[34] := 6; I.ZZCol[34] := 1;
I.ZZRow[35] := 7; I.ZZCol[35] := 0;
I.ZZRow[36] := 7; I.ZZCol[36] := 1;
I.ZZRow[37] := 6; I.ZZCol[37] := 2;
I.ZZRow[38] := 5; I.ZZCol[38] := 3;
I.ZZRow[39] := 4; I.ZZCol[39] := 4;
I.ZZRow[40] := 3; I.ZZCol[40] := 5;
I.ZZRow[41] := 2; I.ZZCol[41] := 6;
I.ZZRow[42] := 1; I.ZZCol[42] := 7;
I.ZZRow[43] := 2; I.ZZCol[43] := 7;
I.ZZRow[44] := 3; I.ZZCol[44] := 6;
I.ZZRow[45] := 4; I.ZZCol[45] := 5;
I.ZZRow[46] := 5; I.ZZCol[46] := 4;
I.ZZRow[47] := 6; I.ZZCol[47] := 3;
I.ZZRow[48] := 7; I.ZZCol[48] := 2;
I.ZZRow[49] := 7; I.ZZCol[49] := 3;
I.ZZRow[50] := 6; I.ZZCol[50] := 4;
I.ZZRow[51] := 5; I.ZZCol[51] := 5;
I.ZZRow[52] := 4; I.ZZCol[52] := 6;
I.ZZRow[53] := 3; I.ZZCol[53] := 7;
I.ZZRow[54] := 4; I.ZZCol[54] := 7;
I.ZZRow[55] := 5; I.ZZCol[55] := 6;
I.ZZRow[56] := 6; I.ZZCol[56] := 5;
I.ZZRow[57] := 7; I.ZZCol[57] := 4;
I.ZZRow[58] := 7; I.ZZCol[58] := 5;
I.ZZRow[59] := 6; I.ZZCol[59] := 6;
I.ZZRow[60] := 5; I.ZZCol[60] := 7;
I.ZZRow[61] := 6; I.ZZCol[61] := 7;
```

```
  I.ZZRow[62] := 7; I.ZZCol[62] := 6;
  I.ZZRow[63] := 7; I.ZZCol[63] := 7;

  writeln('Tables Initialised');
end;

function Fr_Chars (MType : word;Ch : Char) : word;
{* This Function Returns Characteristics About A Frame defined by Ch and   *}
{* Mtype ie # of GOBS or # of MACs per frame.                               *}

begin
  Fr_Chars := 0;
  if Ch = 'M' then
    begin
      if MType = CIF then
        Fr_Chars := MAC_GOB * 12;
      if MType = QCIF then
        Fr_Chars := MAC_GOB * 3;
      if MType = NTSC then
        Fr_Chars := MAC_GOB * 10;
    end;

  if Ch = 'G' then
    begin
      if MType = CIF then
        Fr_Chars := 12;
      if MType = QCIF then
        Fr_Chars := 3;
      if MType = NTSC then
        Fr_Chars := 10;
    end;


end;


Procedure AllocFrame(var Pic : PicType; MovType : byte);
{* This Procedure Allocates working integer space for a frame.             *}

var
  Idx1,Idx2 : word;

begin
  for Idx1 := 1 to Fr_Chars(MovType,'G') do
    begin
      new(Pic[Idx1]);
      Pic[Idx1]^.GQuant := IGQUANT;
    end;
end;

procedure UnloadFrame(var I : Infotype; var Pic : Pictype);
{* This Procedure De-allocates a working Integer frame.                    *}

var
  Idx1,Idx2 : word;

begin
  for Idx1 := 1 to Fr_Chars(I.Mov_Type,'G') do
```

```
    begin
      dispose(Pic[Idx1]);
    end;
end;

Procedure RAllocFrame(var Pic : RpicType; MovType : byte);
{* This Procedure Allocates A Raw Byte Frame To Load-Save A Loose Picture.  *}

var
  Idx1,Idx2 : word;

begin
  for Idx1 := 1 to Fr_Chars(MovType,'G') do
    begin
      new(Pic[Idx1]);
    end;
end;

procedure RUnloadFrame(var I : Infotype; var Pic : RPictype);
{* This Procedure DEAllocates A Raw Byte Frame To Load-Save A Loose Picture.*}

var
  Idx1,Idx2 : word;

begin
  for Idx1 := 1 to Fr_Chars(I.Mov_Type,'G') do
    begin
      dispose(Pic[Idx1]);
    end;
end;



function BSRead(var BS : Bstype) : boolean;
{* This Function Reads A single Byte and places it In the Stream.          *}

var
  InByte : byte;

begin

  if Not BS.FEOF then
    begin
      read(BS.Bsfile,InByte);
      BS.CPos := BS.CPos + 1;

      if BS.CPos = BS.FSize then
        BS.FEOF := True;

      BS.Tr_Bits := BS.Tr_Bits + BtS(InByte);
      BS.VBits := length(BS.Tr_Bits);
      BSRead := True;
    end
  else
    BSRead := false;
end;

function BitOut(var BS : Bstype; var Error : byte) : boolean;
```

```
{* This Function Takes One Bit Out Form the Stream.                *}

var
  Rd : boolean;
  Idx1 : word;
  TmpStr : String;

begin
  Error := 0;
  if (BS.VBits < 1) and (not BS.FEOF) then
    begin
      Error := 0;
      Rd := BSRead(BS);
      if not Rd then
        Error := 1;
    end;
{write(' B:',BS.Tr_Bits);if readkey = ' ' then halt(1);}
  if Error = 0 then
    begin
      if BS.Tr_Bits[1] = '1' then
        BitOut := True
      else
        BitOut := False;
      TmpStr := '';
      For Idx1 := 2 to BS.VBits do
        TmpStr := TmpStr + BS.Tr_Bits[Idx1];

      BS.Tr_Bits := TmpStr;
      BS.VBits := Length (BS.Tr_Bits);
    end;
end;



function BSCmp(var BS : Bstype; var OutStr : string; Pop : boolean) : boolean;
{* This Function Compares BS.Tr_Bits and BS.ChkVal and pops it out to      *}
{* OutStr if POP is activated.                                            *}

var
  Idx1 : word;
  Match : boolean;
  Rd : boolean;
  TmpStr : string;

begin

  BS.CBits := length(BS.ChkVal);
  if BS.CBits < 1 then die ('Cannot Check This',11);

  while (BS.CBits > BS.VBits) and (Not BS.FEOF) do
    begin
      Rd := BSRead(BS);
    end;

  Match := True;

  if BS.VBits >= BS.CBits then
```

```
    begin
      OutStr := '';
      for Idx1 := 1 to BS.CBits do
        begin
          if BS.Tr_Bits[Idx1] <> BS.ChkVal[Idx1] then
            Match := False;
          OutStr := OutStr + BS.Tr_Bits[Idx1];
        end;

      if Pop or Match then
        begin
          TmpStr := '';
          for Idx1 := (BS.CBits+1) to BS.VBits do
            TmpStr := TmpStr + BS.Tr_Bits[Idx1];
          BS.Tr_Bits := TmpStr;
          BS.VBits := length (BS.Tr_Bits);
        end
    end
  else
    Match := False;

  BSCmp := Match;
end;


function ReadPicHeader(var I : Infotype; var BS : Bstype; var Fr_Nu : word) : boolean;
{* This Function Checks for the PSC and gets rid of any Bit Stuffing      *}

var
  Idx1 : word;
  Stuffing : Boolean;
  Match,PSC,Tmp : boolean;
  Residue : string;
  Error : byte;


begin
  ReadPicHeader := True;
  Stuffing := True;
  PSC := False;
  While Stuffing do
    begin
      BS.Chkval := I.MBATab[34];
      Match := BSCmp(BS,Residue,False);

      if Not Match then
        begin
          Stuffing := False;
          BS.Chkval := '00000000000000010000';
          PSC := BSCmp(BS,Residue,False);
        end;
    end;

  ReadPicHeader := PSC;

  if PSC and (Not BS.FEOF) then
    begin
      BS.Chkval := '00000';
```

```
      Match := BSCmp(BS,Residue,True);
      I.Tref := StB(Residue);
      I.Split_Screen := BitOut(BS,Error);
      I.Doc_Cam_Ind := BitOut(BS,Error);
      I.Fr_Pic_Rel := BitOut(BS,Error);

      if BitOut(BS,Error) then
        I.Mov_Type := CIF
      else
        I.Mov_Type := QCIF;

      BS.Chkval := '00';
      Tmp := BSCmp(BS,Residue,True);

      Idx1 := 1;

      while BitOut(BS,Error) do
        begin
          BS.Chkval := '00000000';
          Tmp := BSCmp(BS,Residue,True);
          if (Idx1 = 1) and (Residue = '10001100') then
            I.Mov_Type := NTSC;
          Idx1 := Idx1 + 1;
        end;
    end;

  if Fr_Nu = 2 then
    INFO.DEBUG1 := True
  else
    INFO.DEBUG1 := False;
end;

function SignGet(var BS : Bstype) : Picvaltype;
{* This Function Returns A single Signed value.                    *}

var
  Tmp,Sign : boolean;
  Error : Byte;
  Residue : string;

begin
  Sign := BitOut(BS,Error);
  BS.ChkVal := '0000000';
  Tmp := BSCmp(BS,Residue,True);
  if Sign then
    SignGet := StB(Residue) * -1
  else
    SignGet := StB(Residue);
end;

function GetCoef(var I : Infotype; var BS : Bstype; var Run : byte;
                 var Level : Picvaltype; Fst,RAll : boolean) : boolean;
{* This Function Checks for the correct Run and Levels.            *}

var
  Match,Tmp,Tmp2 : boolean;
  Idx1 : word;
  Residue : string;
```

```
  Error : byte;

begin
  GetCoef := False;
  If RAll then
    begin
      if Fst then
        begin
          Match := False;
          BS.ChkVal := '1';
          Match := BSCmp(BS,Residue,False);
          if Match then
            begin
              Run := 0;
              Level := 1;
              if BitOut(BS,Error) then
                Level := Level * -1;
            end;
        end
      else
        begin
          Match := False;
          BS.ChkVal := '11';
          Match := BSCmp(BS,Residue,False);
          if Match then
            begin
              Run := 0;
              Level := 1;
              if BitOut(BS,Error) then
                Level := Level * -1;
            end;
        end;

      Idx1 := 1;
      while (Idx1 < 64) and (Not Match) do
        begin
          BS.ChkVal := I.COTab[Idx1].COFF;
          Match := BSCmp(BS,Residue,False);
          if not Match then
            Idx1 := Idx1 + 1
          else
            if Idx1 > 1 then
              begin
                Run := I.COTab[Idx1].Run;
                Level := I.COTab[Idx1].Level;
                  if BitOut(BS,Error) then
                    Level := Level * -1;
              end
            else
              begin

                {if INFO.DEBUG1 then}

                BS.ChkVal := '000000';
                Tmp := BSCmp(BS,Residue,True);
                Run := StB(Residue);
                Level := SignGet(BS);
              {  begin
```

```
                write('------ ',Run,' ',Level,' ------');
                if readkey = ' ' then halt(1);
              end;}
            end;
        end;

      if Not Match then
        begin
          BS.ChkVal := '10';
          Match := BSCmp(BS,Residue,False);
          if Match then
            begin
              Run := 0;
              Level := 0;
              GetCoef := True;
            end;

        end;


      if Not Match then
        Die('Coefficient Not Found',17);
    end
  else
    begin
      Match := False;
      BS.ChkVal := '11';
      Match := BSCmp(BS,Residue,False);
      if Match then
        begin
          Run := 0;
          Level := 1;
          if BitOut(BS,Error) then
            Level := Level * -1;
        end;

      Idx1 := 1;
      while (Idx1 < 64) and (Not Match) do
        begin
          BS.ChkVal := I.COTab[Idx1].COFF;
          Match := BSCmp(BS,Residue,False);
          if not Match then
            Idx1 := Idx1 + 1
          else
            if Idx1 > 1 then
              begin
                Run := I.COTab[Idx1].Run;
                Level := I.COTab[Idx1].Level;
                  if BitOut(BS,Error) then
                    Level := Level * -1;
              end
            else
              begin

                BS.ChkVal := '000000';
                Tmp := BSCmp(BS,Residue,True);
                Run := StB(Residue);
                Level := SignGet(BS);
```

```
            end;
        end;

      if Not Match then
        begin
          BS.ChkVal := '10';
          Match := BSCmp(BS,Residue,False);
          if Match then
            begin
              Run := 0;
              Level := 0;
              GetCoef := True;
            end;

        end;

      if Not Match then
        Die('Coefficient Not Found',17);

    end;

end;


procedure BSReadBlk(var I : Infotype; var BS : Bstype; var UMac : Macblkptrtype);
{* This Procedure Reads the Data Coeficients form the Bitstream.          *}

var
  Idx1,Idx2,test : word;
  Run : byte;
  Level : Picvaltype;
  Fst : boolean;
  EOB : boolean;
  Ln1,Ln2,Ln3,Ln4,Ln5,Ln6 : Blklinetype;
  Row,Col : byte;



begin
  if UMac^.MType in [1,2] then
    begin
      write('~');
      EOB := False;

      Ln1[1] := SignGet(BS);
      Idx1 := 2;
      while (Not EOB) do
        begin
          EOB := GetCoef(I,BS,Run,Level,False,False);
          if Not EOB then
            begin
              if Run <> 0 then
                begin
                  for Idx2 := 0 to Run-1 do
                    begin
                      Ln1[Idx1] := 0;
                      Idx1 := Idx1 + 1;
                    end
```

```
          end;
        Ln1[Idx1] := Level;
        Idx1 := Idx1 + 1;
      end
    else
      begin
        while Idx1 < (BLK_WID*BLK_HIG + 1) do
          begin
            Ln1[Idx1] := 0;
            Idx1 := Idx1 + 1;
          end;
      end;
  end;


EOB := False;
Ln2[1] := SignGet(BS);
Idx1 := 2;
while (Not EOB) do
  begin
    EOB := GetCoef(I,BS,Run,Level,False,False);


    if Not EOB then
      begin
        if Run <> 0 then
          begin
            for Idx2 := 0 to Run-1 do
              begin
                Ln2[Idx1] := 0;
                Idx1 := Idx1 + 1;
              end
          end;
        Ln2[Idx1] := Level;
        Idx1 := Idx1 + 1;
      end
    else
      begin
        while Idx1 < (BLK_WID*BLK_HIG + 1) do
          begin
            Ln2[Idx1] := 0;
            Idx1 := Idx1 + 1;
          end;
      end;
  end;

EOB := False;
Ln3[1] := SignGet(BS);
Idx1 := 2;
while (Not EOB) do
  begin
    EOB := GetCoef(I,BS,Run,Level,False,False);
    if Not EOB then
      begin
        if Run <> 0 then
          begin

            for Idx2 := 0 to Run-1 do
```

```
            begin
              Ln3[Idx1] := 0;
              Idx1 := Idx1 + 1;
            end
        end;
      Ln3[Idx1] := Level;
      Idx1 := Idx1 + 1;
    end
  else
    begin
      while Idx1 < (BLK_WID*BLK_HIG + 1) do
        begin
          Ln3[Idx1] := 0;
          Idx1 := Idx1 + 1;
        end;
    end;
end;

EOB := False;
Ln4[1] := SignGet(BS);
Idx1 := 2;
while (Not EOB) do
  begin
    EOB := GetCoef(I,BS,Run,Level,False,False);
    if Not EOB then
      begin
        if Run <> 0 then
          begin
            for Idx2 := 0 to Run-1 do
              begin
                Ln4[Idx1] := 0;
                Idx1 := Idx1 + 1;
              end
          end;
        Ln4[Idx1] := Level;
        Idx1 := Idx1 + 1;
      end
    else
      begin
        while Idx1 < (BLK_WID*BLK_HIG + 1) do
          begin
            Ln4[Idx1] := 0;
            Idx1 := Idx1 + 1;
          end;
      end;
  end;

EOB := False;
Ln5[1] := SignGet(BS);
Idx1 := 2;
while (Not EOB) do
  begin
    EOB := GetCoef(I,BS,Run,Level,False,False);
    if Not EOB then
      begin
        if Run <> 0 then
          begin
            for Idx2 := 0 to Run-1 do
```

```
                begin
                  Ln5[Idx1] := 0;
                  Idx1 := Idx1 + 1;
                end
            end;
          Ln5[Idx1] := Level;
          Idx1 := Idx1 + 1;
        end
      else
        begin
          while Idx1 < (BLK_WID*BLK_HIG + 1) do
            begin
              Ln5[Idx1] := 0;
              Idx1 := Idx1 + 1;
            end;
        end;
    end;

EOB := False;
Ln6[1] := SignGet(BS);
Idx1 := 2;
while (Not EOB) do
  begin
    EOB := GetCoef(I,BS,Run,Level,False,False);
    if Not EOB then
      begin
        if Run <> 0 then
          begin
            for Idx2 := 0 to Run-1 do
              begin
                Ln6[Idx1] := 0;
                Idx1 := Idx1 + 1;
              end
          end;
        Ln6[Idx1] := Level;
        Idx1 := Idx1 + 1;
      end
    else
      begin
        while Idx1 < (BLK_WID*BLK_HIG + 1) do
          begin
            Ln6[Idx1] := 0;
            Idx1 := Idx1 + 1;
          end;
      end;
  end;


for Idx1 := 2 to BLK_WID*BLK_HIG do
  begin
    Row := I.ZZRow[Idx1-1];
    Col := I.ZZCol[Idx1-1];

    UMac^.RawmacYArray[Row+1,Col+1] := Ln1[Idx1];
    UMac^.RawmacYArray[Row+1,Col+BLK_WID+1] := Ln2[Idx1];
    UMac^.RawmacYArray[Row+BLK_HIG+1,Col+1] := Ln3[Idx1];
    UMac^.RawmacYArray[Row+BLK_HIG+1,Col+BLK_WID+1] := Ln4[Idx1];
    UMac^.RawmacUArray[Row+1,Col+1] := Ln5[Idx1];
```

```
        UMac^.RawmacVArray[Row+1,Col+1] := Ln6[Idx1];

      end;

    UMac^.RawmacYArray[1,1] := Ln1[1];
    UMac^.RawmacYArray[1,BLK_WID + 1] := Ln2[1];
    UMac^.RawmacYArray[BLK_HIG+1,1] := Ln3[1];
    UMac^.RawmacYArray[BLK_HIG+1,BLK_WID+1] := Ln4[1];
    UMac^.RawmacUArray[1,1] := Ln5[1];
    UMac^.RawmacVArray[1,1] := Ln6[1];
  end
else
  begin
    write('+');

    EOB := False;
    Idx1 := 1;
    Fst := True;
    if ((UMac^.CBP and 32) = 32) then
      begin
        while (Not EOB) do
          begin
            EOB := GetCoef(I,BS,Run,Level,Fst,True);
            Fst := False;
            if Not EOB then
              begin
                if Run <> 0 then
                  begin
                    for Idx2 := 0 to Run-1 do
                      begin
                        Ln1[Idx1] := 0;
                        Idx1 := Idx1 + 1;
                      end
                  end;
                Ln1[Idx1] := Level;
                Idx1 := Idx1 + 1;
              end
            else
              begin
                while Idx1 < (BLK_WID*BLK_HIG + 1) do
                  begin
                    Ln1[Idx1] := 0;
                    Idx1 := Idx1 + 1;
                  end;
              end;
          end;
      end
    else
      begin
        for Run := 1 to BLK_WID*BLK_HIG do
          Ln1[Run] := 0;
      end;


    EOB := False;
    Idx1 := 1;
    Fst := True;
    if ((UMac^.CBP and 16) = 16) then
```

```
        begin
          while (Not EOB) do
            begin
              EOB := GetCoef(I,BS,Run,Level,Fst,True);
              Fst := False;
              if Not EOB then
                begin
                  if Run <> 0 then
                    begin
                      for Idx2 := 0 to Run-1 do
                        begin
                          Ln2[Idx1] := 0;
                          Idx1 := Idx1 + 1;
                        end
                    end;
                  Ln2[Idx1] := Level;
                  Idx1 := Idx1 + 1;
                end
              else
                begin
                  while Idx1 < (BLK_WID*BLK_HIG + 1) do
                    begin
                      Ln2[Idx1] := 0;
                      Idx1 := Idx1 + 1;
                    end;
                end;
            end;
        end
      else
        begin
          for Run := 1 to BLK_WID*BLK_HIG do
            Ln2[Run] := 0;
        end;

{write('B');}

      EOB := False;
      Idx1 := 1;
      Fst := True;
      if ((UMac^.CBP and 8) = 8) then
        begin
          while (Not EOB) do
            begin
              EOB := GetCoef(I,BS,Run,Level,Fst,True);
              Fst := False;
              {write(' I:',Idx1,' R:',Run,' L:',Level);}
              if Not EOB then
                begin
                  if Run <> 0 then
                    begin
                      for Idx2 := 0 to Run-1 do
                        begin

                          Ln3[Idx1] := 0;
                          Idx1 := Idx1 + 1;
                        end
                    end;
                  Ln3[Idx1] := Level;
```

```
                Idx1 := Idx1 + 1;
              end
            else
              begin
                while Idx1 < (BLK_WID*BLK_HIG + 1) do
                  begin
                    Ln3[Idx1] := 0;
                    Idx1 := Idx1 + 1;
                  end;
              end;
          end;
      end
    else
      begin
        for Run := 1 to BLK_WID*BLK_HIG do
          Ln3[Run] := 0;
      end;
{for Idx1 := 1 to 64 do
      write(Ln3[Idx1],',');
    if readkey = ' ' then halt(1);}


    EOB := False;
    Idx1 := 1;
    Fst := True;
    if ((UMac^.CBP and 4) = 4) then
      begin
        while (Not EOB) do
          begin
            EOB := GetCoef(I,BS,Run,Level,Fst,True);
            Fst := False;
            if Not EOB then
              begin
                if Run <> 0 then
                  begin
                    for Idx2 := 0 to Run-1 do
                      begin
                        Ln4[Idx1] := 0;
                        Idx1 := Idx1 + 1;
                      end
                  end;
                Ln4[Idx1] := Level;
                Idx1 := Idx1 + 1;
              end
            else
              begin
                while Idx1 < (BLK_WID*BLK_HIG + 1) do
                  begin
                    Ln4[Idx1] := 0;
                    Idx1 := Idx1 + 1;
                  end;
              end;
          end;
      end
    else
      begin
        for Run := 1 to BLK_WID*BLK_HIG do
          Ln4[Run] := 0;
```

```
  end;

EOB := False;
Idx1 := 1;
Fst := True;
if ((UMac^.CBP and 2) = 2) then
  begin
    while (Not EOB) do
      begin
        EOB := GetCoef(I,BS,Run,Level,Fst,True);
        Fst := False;
        if Not EOB then
          begin
            if Run <> 0 then
              begin
                for Idx2 := 0 to Run-1 do
                  begin
                    Ln5[Idx1] := 0;
                    Idx1 := Idx1 + 1;
                  end
              end;
            Ln5[Idx1] := Level;
            Idx1 := Idx1 + 1;
          end
        else
          begin
            while Idx1 < (BLK_WID*BLK_HIG + 1) do
              begin
                Ln5[Idx1] := 0;
                Idx1 := Idx1 + 1;
              end;
          end;
      end;
  end
else
  begin
    for Run := 1 to BLK_WID*BLK_HIG do
      Ln5[Run] := 0;
  end;

EOB := False;
Idx1 := 1;
Fst := True;
if ((UMac^.CBP and 1) = 1) then
  begin
    while (Not EOB) do
      begin
        EOB := GetCoef(I,BS,Run,Level,Fst,True);
        Fst := False;
        if Not EOB then
          begin
            if Run <> 0 then
              begin
                for Idx2 := 0 to Run-1 do
                  begin
                    Ln6[Idx1] := 0;
                    Idx1 := Idx1 + 1;
                  end
```

```
                end;
              Ln6[Idx1] := Level;
              Idx1 := Idx1 + 1;
            end
          else
            begin
              while Idx1 < (BLK_WID*BLK_HIG + 1) do
                begin
                  Ln6[Idx1] := 0;
                  Idx1 := Idx1 + 1;
                end;
            end;
        end;
    end
  else
    begin
      for Run := 1 to BLK_WID*BLK_HIG do
        Ln6[Run] := 0;
    end;

  for Idx1 := 2 to BLK_WID*BLK_HIG do
    begin
      Row := I.ZZRow[Idx1-1];
      Col := I.ZZCol[Idx1-1];

      UMac^.RawmacYArray[Row+1,Col+1] := Ln1[Idx1];
      UMac^.RawmacYArray[Row+1,Col+BLK_WID+1] := Ln2[Idx1];
      UMac^.RawmacYArray[Row+BLK_HIG+1,Col+1] := Ln3[Idx1];
      UMac^.RawmacYArray[Row+BLK_HIG+1,Col+BLK_WID+1] := Ln4[Idx1];
      UMac^.RawmacUArray[Row+1,Col+1] := Ln5[Idx1];
      UMac^.RawmacVArray[Row+1,Col+1] := Ln6[Idx1];

    end;

  UMac^.RawmacYArray[1,1] := Ln1[1];
  UMac^.RawmacYArray[1,BLK_WID + 1] := Ln2[1];
  UMac^.RawmacYArray[BLK_HIG+1,1] := Ln3[1];
  UMac^.RawmacYArray[BLK_HIG+1,BLK_WID+1] := Ln4[1];
  UMac^.RawmacUArray[1,1] := Ln5[1];
  UMac^.RawmacVArray[1,1] := Ln6[1];
end;


end;



procedure BSFrameLoad(var I : Infotype; var BS : Bstype; var UPic : Pictype);
{* This Procedure Loads the Frame From the Bitstream.                    *}

var
  GOB,MAC,GNum,MNum,MDiff : byte;
  Gob_Num,Error : byte;
  Idx1 : word;
  GSC,Tmp,Match : boolean;
  Residue : string;
  MVDH,MVDV,MVD1,MVD2 : integer;
  UMac : Macblkptrtype;
```

Appendix B

```
begin
  Gob_Num := Fr_Chars(I.Mov_Type,'G');
  for GOB := 1 to Gob_Num do
    begin
      BS.ChkVal := '0000000000000001';
      GSC := BSCmp(BS,Residue,False);

      if Not GSC then
        Die('GSC Not Found',13);

      BS.ChkVal := '0000';
      GSC := BSCmp(BS,Residue,True);
      GNum := StB(Residue);

      BS.ChkVal := '00000';
      GSC := BSCmp(BS,Residue,True);
      UPic[GNum]^.GQuant := StB(Residue);

      while BitOut(BS,Error) do
        begin
          BS.Chkval := '00000000';
          Tmp := BSCmp(BS,Residue,True);
        end;

      MNum := 0;
      for MAC := 1 to MAC_GOB do
        begin
          Idx1 := 1;
          Match := False;

          while (Idx1 < 35) and (not Match) do
            begin
              BS.Chkval := I.MBATab[Idx1];
              Match := BSCmp(BS,Residue,False);

              If (Idx1 = 34) and Match then
                begin
                  Idx1 := 0;
                  Match := False;
                end;

              if Not Match then
                Idx1 := Idx1 + 1;
            end;


          if Not Match then
            Die('Cannot Find Macroblock Number',14);

          MDiff := Idx1;

          MNum := MNum + Idx1;


          Idx1 := 1;
          Match := false;
```

```
while (Idx1 < 11) and (Not Match) do
  begin
    BS.Chkval := I.MTypeTab[Idx1];
    Match := BSCmp(BS,Residue,False);
    if Not Match then
      Idx1 := Idx1 + 1;
  end;


if not Match then
  die('Cannot Find MType',15);

UPic[GNum]^.Macs[MNum].Mtype := Idx1;



if UPic[GNum]^.Macs[MNum].Mtype in [2,4,7,10] then
  begin
    BS.Chkval := '00000';
    Match := BSCmp(BS,Residue,True);
    UPic[GNum]^.Macs[MNum].MQuant := StB(Residue);
  end;


if UPic[GNum]^.Macs[MNum].Mtype in [5..10] then
  begin
    Idx1 := 0;
    Match := false;
    while (Idx1 < 32) and (Not Match) do
      begin
        BS.Chkval := I.MVDTab[Idx1];
        Match := BSCmp(BS,Residue,False);
        if not Match then
          Idx1 := Idx1 + 1;
      end;

    if (Not Match) then
      Die('MVDH cannot be found',16);


    MVDH := Idx1;

    Idx1 := 0;
    Match := false;
    while (Idx1 < 32) and (Not Match) do
      begin
        BS.Chkval := I.MVDTab[Idx1];
        Match := BSCmp(BS,Residue,False);
        if not Match then
          Idx1 := Idx1 + 1;
      end;

    if (Not Match) then
      Die('MVDV cannot be found',16);

    MVDV := Idx1;
```

```
    if (MNum in [1,12,23]) then
      begin
        if MVDH > 15 then
          UPic[GNum]^.Macs[MNum].MVH := MVDH - 32
        else
          UPic[GNum]^.Macs[MNum].MVH := MVDH;

        if MVDV > 15 then
          UPic[GNum]^.Macs[MNum].MVV := MVDV - 32
        else
          UPic[GNum]^.Macs[MNum].MVV := MVDV;
      end
    else
      begin
        if (UPic[GNum]^.Macs[MNum-1].MType in [1..4]) or (MDiff <> 1) then
          begin
            if MVDH > 15 then
              UPic[GNum]^.Macs[MNum].MVH := MVDH - 32
            else
              UPic[GNum]^.Macs[MNum].MVH := MVDH;

            if MVDV > 15 then
              UPic[GNum]^.Macs[MNum].MVV := MVDV - 32
            else
              UPic[GNum]^.Macs[MNum].MVV := MVDV;
          end
        else
          begin
            MVD1 := UPic[GNum]^.Macs[MNum-1].MVH + MVDH;
            if (MVD1 > -16) and (MVD1 < 16) then
              UPic[GNum]^.Macs[MNum].MVH :=
               UPic[GNum]^.Macs[MNum-1].MVH + MVDH
            else
              UPic[GNum]^.Macs[MNum].MVH :=
               UPic[GNum]^.Macs[MNum-1].MVH + MVDH - 32;

            MVD1 := UPic[GNum]^.Macs[MNum-1].MVV + MVDV;
            if (MVD1 > -16) and (MVD1 < 16) then
              UPic[GNum]^.Macs[MNum].MVV :=
               UPic[GNum]^.Macs[MNum-1].MVV + MVDV
            else
              UPic[GNum]^.Macs[MNum].MVV :=
               UPic[GNum]^.Macs[MNum-1].MVV + MVDV - 32;
          end;
      end;
  end;

if UPic[GNum]^.Macs[MNum].MType in [3,4,6,7,9,10] then
  begin
    Idx1 := 1;
    Match := False;

    while (Idx1 < 64) and (Not Match) do
      begin
        BS.Chkval := I.CBPTab[Idx1];
        Match := BSCmp(BS,Residue,False);
        if not Match then
```

```
                Idx1 := Idx1 + 1;
              end;

            if (Not Match) then
              Die('CBP cannot be found',17);

            UPic[GNum]^.Macs[MNum].CBP := Idx1;
          end;


        if UPic[GNum]^.Macs[MNum].MType in [1,2,3,4,6,7,9,10] then
          begin
            Umac := @((UPic[GNum])^.Macs[MNum]);
            BSReadBlk(1,BS,UMac);
          end;
      end;

  end;

end;

procedure ConvToM_G(Mov_Type : byte;RawMAC : word; var GOB,MAC : word);
{* Returns GOB and MAC of a given Raw Macroblock.                    *}

var
  Idx1 : word;
  MACRow,MACCol : word;

begin
  If Mov_Type in [NTSC,CIF] then
    begin
      MACRow := ((RawMAC - 1) div (GOB_WID * 2)) + 1;
      MACCol := ((RawMAC - 1) mod (GOB_WID * 2)) + 1;

      if MACCol > GOB_WID then
        begin
          MACCol := MACCol - GOB_WID;
          MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACCol;
          GOB := (((MacRow - 1) div GOB_HIG) + 1) * 2;
        end

      else
        begin
          MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACCol;
          GOB := ((((MacRow - 1) div GOB_HIG) + 1) * 2) - 1;
        end;
    end
  else
    If Mov_Type in [QCIF] then
      begin
        MACRow := ((RawMAC - 1) div (GOB_WID)) + 1;
        MACCol := ((RawMAC - 1) mod (GOB_WID)) + 1;
        MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACCol;
        GOB := (((MacRow - 1) div GOB_HIG) + 1);     { * 2) - 1;}
      end;

end;
```

Appendix B

```
procedure ConMCRData(Num : word; var MAC,Row,Col : byte; SBlk : boolean);
{* This Function Returns ther mac and a 2D Pointer to a value in the mac    *}

var
  Row1,Col1,MACRow,MACCol : word;
  BlkWid,BlkHig : word;

begin
  if SBlk then
    begin
      BlkWid := BLK_WID*2;
      BlkHig := BLK_HIG*2;
    end
  else
    begin
      BlkWid := BLK_WID;
      BlkHig := BLK_HIG;
    end;

  Row1 := (Num - 1) div (3 * BlkWid) + 1;
  Col1 := (Num - 1) mod (3 * BlkWid) + 1;
  MACRow := ((Row1 - 1) div BlkHig) + 1;
  MACCol := ((Col1 - 1) div BlkWid) + 1;
  Row := ((Row1 - 1) mod BlkHig) + 1;
  Col := ((Col1 - 1) mod BlkWid) + 1;

  MAC := (MacRow * 3) + MACCol - 3;          {* Hmmm Interesting Discovery *}

end;


function DeQuantise (Value : Picvaltype; Quant : byte) : DCTvaltype;
{* Performs De quantisation According to CCITT standard.                    *}

var
  Tmp : DCTvaltype;

begin
  if Value = 0 then
    DeQuantise := 0
  else
    begin
      if (Quant mod 2) = 1 then
        begin
          if Value > 0 then
            Tmp := (2*Value+1)*Quant
          else
            Tmp := (2*Value-1)*Quant;
        end
      else
        begin
          if Value > 0 then
            Tmp := (2*Value+1)*Quant - 1
          else
            Tmp := (2*Value-1)*Quant + 1;
        end;
```

```
      DeQuantise := Tmp;
    end;
end;

function Clip2(Value : DCTvaltype) : Rawpicvaltype;
{* Performs CCITT Clipping On For Raw Pictur Format after DCT and MC.      *}

var
 Tmp : integer;

begin
      Tmp := round(Value);

      if Tmp > 255 then
        Clip2 := 255
      else
        if Tmp < 0 then
          Clip2 := 0
        else
          Clip2 := Tmp;
end;


function Clip(Value : DCTvaltype) : Picvaltype;
{* Performs CCITT Clipping for Values for work frame after DCT and Quantise *}

begin
      Clip := round(Value);

      if Value > 255 then
        Clip := 255;

      if Value < -256 then
        Clip := -256;
end;

procedure DEQ_IDCT(var UMac : Macblkptrtype; Quant : byte; GQuant : boolean);
{* This Procedure Does The IDCT and calls De Quantisation.                *}

var
  Mult,Mult2,Calc : DCTvaltype;
  vSum1,uSum1 : DCTvaltype;
  vSum2,uSum2 : DCTvaltype;
  vSum3,uSum3 : DCTvaltype;
  vSum4,uSum4 : DCTvaltype;
  vSum5,uSum5 : DCTvaltype;
  vSum6,uSum6 : DCTvaltype;
  u,v,i,j,DSiz : byte;
  NMac : MacBlkptrtype;

begin
  Mult := 1/sqrt(2);
  Mult2 := 1/4;
  DSiz := BLK_WID + BLK_HIG;
  new (NMac);

  with UMac^ do
```

```
for i := 0 to (BLK_HIG - 1) do
  begin
    for j := 0 to (BLK_WID - 1) do
      begin
        uSum1 := 0;
        uSum2 := 0;
        uSum3 := 0;
        uSum4 := 0;
        uSum5 := 0;
        uSum6 := 0;
        for u := 0 to (BLK_HIG - 1) do
          begin
            vSum1 := 0;
            vSum2 := 0;
            vSum3 := 0;
            vSum4 := 0;
            vSum5 := 0;
            vSum6 := 0;
            for v := 0 to (BLK_WID - 1) do
              begin
                Calc := cos((2*i + 1)*u*pi/DSiz) *
                     cos((2*j + 1)*v*pi/DSiz);

                if v = 0 then
                  Calc := Calc * Mult;

                if u = 0 then
                  Calc := Calc * Mult;

                vSum1 := vSum1 + (DeQuantise(RawMacYArray[(u+1),(v+1)],Quant) * Calc);
                vSum2 := vSum2 + (DeQuantise(RawMacYArray[(u+1),(v+1+BLK_WID)],Quant) *
Calc);
                vSum3 := vSum3 + (DeQuantise(RawMacYArray[(u+1+BLK_HIG),(v+1)],Quant) *
Calc);
                vSum4 := vSum4 +
(DeQuantise(RawMacYArray[(u+1+BLK_HIG),(v+1+BLK_WID)],Quant) * Calc);

                vSum5 := vSum5 + (DeQuantise(RawMacUArray[(u+1),(v+1)],Quant) * Calc);
                vSum6 := vSum6 + (DeQuantise(RawMacVArray[(u+1),(v+1)],Quant) * Calc);
              end;

            uSum1 := uSum1 + vSum1;
            uSum2 := uSum2 + vSum2;
            uSum3 := uSum3 + vSum3;
            uSum4 := uSum4 + vSum4;
            uSum5 := uSum5 + vSum5;
            uSum6 := uSum6 + vSum6;

          end;


        uSum1 := uSum1 * Mult2;
        uSum2 := uSum2 * Mult2;
        uSum3 := uSum3 * Mult2;
        uSum4 := uSum4 * Mult2;
        uSum5 := uSum5 * Mult2;
        uSum6 := uSum6 * Mult2;
```

```
            NMac^.RawMacYArray[i+1,j+1] := Clip(uSum1);
            NMac^.RawMacYArray[i+1,j+1+BLK_WID] := Clip(uSum2);
            NMac^.RawMacYArray[i+1+BLK_HIG,j+1] := Clip(uSum3);
            NMac^.RawMacYArray[i+1+BLK_HIG,j+1+BLK_WID] := Clip(uSum4);
            NMac^.RawMacUArray[i+1,j+1] := Clip(uSum5);
            NMac^.RawMacVArray[i+1,j+1] := Clip(uSum6);
          end;
    end;

  UMac^.RawMacYArray := NMac^.RawMacYArray;
  UMac^.RawMacUArray := NMac^.RawMacUArray;
  UMac^.RawMacVArray := NMac^.RawMacVArray;
  dispose(NMac);
end;


procedure DeQuant_IDCT(var I : Infotype; var UPic : Pictype);
{* This procedure Calls IDCT and De Quantise for every macro block.        *}

var
  GOB,MAC : byte;
  Idx1,Idx2 : word;
  CMac : Macblkptrtype;
  MCA : Mcatype;
  Top,Bottom,Left,Right : boolean;

begin
  for GOB := 1 to Fr_Chars(I.Mov_Type,'G') do
    begin
      For MAC := 1 to MAC_GOB do
        begin
          write(':');
          CMac := @(UPic[GOB]^.Macs[MAC]);
          DEQ_IDCT(CMac,UPic[GOB]^.GQuant,True);
        end;
    end;
end;

function DeMCSearch(var MCA : Mcatype; var UMac : Macblkptrtype; var CMac : RMacptrtype) :
boolean;
{* This Function Undoes the search done by MC Search to give the DCT and    *}
{* Quantisation Altered value to use as the previous picture.          *}

var
  Pos : word;
  MAC,Row,Col : byte;
  Idx1,Idx2 : byte;

begin
  DeMCSearch := False;

  if UMac^.MType in [1..2] then
    begin
      write(UMac^.MType);
      DeMCSearch := True;
```

```
      for Row := 1 to (BLK_HIG*2) do
        for Col := 1 to (BLK_WID*2) do
          begin
            CMac^.RawMacYArray[Row,Col] := Clip2(UMac^.RawMacYArray[Row,Col]);
            if ((Row mod 2) = 0) and ((Col mod 2) = 0) then
              begin
                CMac^.RawMacUArray[Row div 2,Col div 2] :=
                Clip2(UMac^.RawMacUArray[Row div 2,Col div 2]);
                CMac^.RawMacVArray[Row div 2,Col div 2] :=
                Clip2(UMac^.RawMacVArray[Row div 2,Col div 2]);
              end;
          end;
    end;

  if UMac^.MType in [3..4] then
    begin
      write(UMac^.MType);
      for Row := 1 to (BLK_HIG) do
        for Col := 1 to (BLK_WID) do
          begin
            if (UMac^.CBP and 32) = 32 then
              begin
                CMac^.RawMacYArray[Row,Col] :=
                Clip2(UMac^.RawMacYArray[Row,Col] + MCA[5]^.RawMacYArray[Row,Col]/1);
              end
            else
              begin
                CMac^.RawMacYArray[Row,Col] := Clip2(MCA[5]^.RawMacYArray[Row,Col]/1);
              end;

            if (UMac^.CBP and 16) = 16 then
              begin
                CMac^.RawMacYArray[Row,Col+BLK_WID] :=
                Clip2(UMac^.RawMacYArray[Row,Col+BLK_WID]
                + MCA[5]^.RawMacYArray[Row,Col+BLK_WID]/1);
              end
            else
              begin
                CMac^.RawMacYArray[Row,Col+BLK_WID] :=
Clip2(MCA[5]^.RawMacYArray[Row,Col+BLK_WID]/1);
              end;

            if (UMac^.CBP and 8) = 8 then
              begin
                CMac^.RawMacYArray[Row+BLK_HIG,Col] :=
                Clip2(UMac^.RawMacYArray[Row+BLK_HIG,Col]
                + MCA[5]^.RawMacYArray[Row+BLK_HIG,Col]/1);
              end
            else
              begin
                CMac^.RawMacYArray[Row+BLK_HIG,Col] :=
Clip2(MCA[5]^.RawMacYArray[Row+BLK_HIG,Col]/1);
              end;

            if (UMac^.CBP and 4) = 4 then
              begin
                CMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID] :=
                Clip2(UMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]
```

```
            + MCA[5]^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]/1);
          end
        else
          begin
            CMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID] :=
            Clip2(MCA[5]^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]/1);
          end;

        if (UMac^.CBP and 2) = 2 then
          begin
            CMac^.RawMacUArray[Row,Col] :=
            Clip2(UMac^.RawMacUArray[Row,Col]
            + MCA[5]^.RawMacUArray[Row,Col]/1);
          end
        else
          begin
            CMac^.RawMacUArray[Row,Col] :=
            Clip2(MCA[5]^.RawMacUArray[Row,Col]/1);
          end;

        if (UMac^.CBP and 1) = 1 then
          begin
            CMac^.RawMacVArray[Row,Col] :=
            Clip2(UMac^.RawMacVArray[Row,Col]
            + MCA[5]^.RawMacVArray[Row,Col]/1);
          end
        else
          begin
            CMac^.RawMacVArray[Row,Col] :=
            Clip2(MCA[5]^.RawMacVArray[Row,Col]/1);
          end;
      end;


  end;

if UMac^.MType in [5,8] then
  begin
    write(UMac^.MType);
    for Row := 1 to (BLK_HIG*2) do
      for Col := 1 to (BLK_WID*2) do
        begin
          Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
            + (BLK_WID * 2 * (Row-1) * 3) + (Col-1);

          ConMCRData(Pos,MAC,Idx1,Idx2,True);

          CMac^.RawMacYArray[Row,Col] := Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]/1);
        end;

    for Row := 1 to (BLK_HIG) do
      for Col := 1 to (BLK_WID) do
        begin
          Pos := ZRO_POS1 + round(UMac^.MVH/2) + (3 * BLK_WID * round(UMac^.MVV/2))
            + (BLK_WID * (Row-1) * 3) + (Col-1);

          ConMCRData(Pos,MAC,Idx1,Idx2,False);
```

```
            CMac^.RawMacUArray[Row,Col] := Clip2(MCA[MAC]^.RawMacUArray[Idx1,Idx2]/1);
            CMac^.RawMacVArray[Row,Col] := Clip2(MCA[MAC]^.RawMacVArray[Idx1,Idx2]/1);
          end;
    end;

  if UMac^.MType in [6,7,9,10] then
    begin
      write(UMac^.MType);
      for Row := 1 to (BLK_HIG) do
        for Col := 1 to (BLK_WID) do
          begin

            Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
                + (BLK_WID * 2 * (Row-1) * 3) + (Col-1);
            ConMCRData(Pos,MAC,Idx1,Idx2,True);

            if (UMac^.CBP and 32) = 32 then
              begin
                CMac^.RawMacYArray[Row,Col] := Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]
                + UMac^.RawMacYArray[Row,Col]/1);
              end
            else
              begin
                CMac^.RawMacYArray[Row,Col] := Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]);
              end;

            Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
                + (BLK_WID * 2 * (Row-1) * 3) + (Col-1+BLK_WID);
            ConMCRData(Pos,MAC,Idx1,Idx2,True);

            if (UMac^.CBP and 16) = 16 then
              begin
                CMac^.RawMacYArray[Row,Col+BLK_WID] :=
Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]
                + UMac^.RawMacYArray[Row,Col+BLK_WID]/1);
              end
            else
              begin
                CMac^.RawMacYArray[Row,Col+BLK_WID] :=
Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]);
              end;

            Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
                + (BLK_WID * 2 * (Row-1+BLK_HIG) * 3) + (Col-1);
            ConMCRData(Pos,MAC,Idx1,Idx2,True);

            if (UMac^.CBP and 8) = 8 then
              begin
                CMac^.RawMacYArray[Row+BLK_HIG,Col] :=
Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]
                + UMac^.RawMacYArray[Row+BLK_HIG,Col]/1);
              end
            else
              begin
                CMac^.RawMacYArray[Row+BLK_HIG,Col] :=
Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]);
              end;
```

```
            Pos := ZRO_POS + UMac^.MVH + (3 * BLK_WID * 2 * UMac^.MVV)
                + (BLK_WID * 2 * (Row-1+BLK_HIG) * 3) + (Col-1+BLK_WID);
            ConMCRData(Pos,MAC,Idx1,Idx2,True);

            if (UMac^.CBP and 4) = 4 then
              begin
                CMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID] :=
                Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]
                + UMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID]/1);
              end
            else
              begin
                CMac^.RawMacYArray[Row+BLK_HIG,Col+BLK_WID] :=
                Clip2(MCA[MAC]^.RawMacYArray[Idx1,Idx2]);
              end;

            Pos := ZRO_POS1 + round(UMac^.MVH/2) + (3 * BLK_WID * round(UMac^.MVV/2))
                + (BLK_WID * (Row-1) * 3) + (Col-1);

            ConMCRData(Pos,MAC,Idx1,Idx2,False);

            if (UMac^.CBP and 2) = 2 then
              begin
                CMac^.RawMacUArray[Row,Col] := Clip2(MCA[MAC]^.RawMacUArray[Idx1,Idx2]
                  + UMac^.RawMacUArray[Row,Col]/1);
              end
            else
              begin
                CMac^.RawMacUArray[Row,Col] := Clip2(MCA[MAC]^.RawMacUArray[Idx1,Idx2]);
              end;

            if (UMac^.CBP and 1) = 1 then
              begin
                CMac^.RawMacVArray[Row,Col] := Clip2(MCA[MAC]^.RawMacVArray[Idx1,Idx2]
                  + UMac^.RawMacVArray[Row,Col]/1);
              end
            else
              begin
                CMac^.RawMacVArray[Row,Col] := Clip2(MCA[MAC]^.RawMacVArray[Idx1,Idx2]);
              end;
          end;
      end;
end;


procedure DeMotion (var I : Infotype; var UPic : Pictype;var CPic,PPic : Rpictype; Pfe : boolean);
{* This Procedure Provides the surrounding blocks for the previous picture  *}
{* to Undo the Motion Search.                                               *}

var
  Idx1,Idx2 : word;
  GOB,MAC,GOB1,MAC1,MAC_Num : word;
  MCA : Mcatype;
  Top,Bottom,Left,Right : boolean;
  UMac : Macblkptrtype;
  Intra : boolean;
  CMac : Rmacptrtype;
```

Appendix B

```
begin
  if not (I.Mov_Type in [CIF,QCIF,NTSC]) then
    Die ('Illigal Picture type (MotionAnalysis)',3);

  MAC_Num := Fr_Chars(I.Mov_Type,'M');
  for Idx1 := 1 to MAC_Num do
    begin
      ConvToM_G(I.Mov_Type,Idx1,GOB,MAC);

      UMac := @((UPic[GOB])^.Macs[MAC]);
      CMac := @((CPic[GOB])^.Macs[MAC]);

      if I.Mov_Type = CIF then
        begin
          If (GOB mod 2) = 1 then
            begin
              if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
                begin
                  Left := False;
                end
              else
                begin
                  Left := True;
                end;
            end
          else
            begin
              if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
                    begin
                      Right := False;
                    end
                  else
                    begin
                      Right := True;
                    end;
                end;

              If GOB in [1,2] then
                begin
                  if MAC in [1..GOB_WID] then
                    begin
                      Top := False;
                    end
                  else
                    begin
                      Top := True;
                    end;
                end;

              If GOB in [11,12] then
                begin
                  if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
                    begin
                      Bottom := False;
                    end
                  else
                    begin
```

```
            Bottom := True;
          end;
      end;
    if not (GOB in [1,2,11,12]) then
      begin
        Top := True;
        Bottom := True;
      end;
  end;

if I.Mov_Type = QCIF then
  begin
    if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
      begin
        Left := False;
      end
    else
      begin
        Left := True;
      end;

    if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
      begin
        Right := False;
      end
    else
      begin
        Right := True;
      end;

    If GOB = 1 then
      begin
        if MAC in [1..GOB_WID] then
          begin
            Top := False;
          end
        else
          begin
            Top := True;
          end;
      end;

    If GOB = 3 then
      begin
        if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
          begin
            Bottom := False;
          end
        else
          begin
            Bottom := True;
          end;
      end;

    if not (GOB in [1,3]) then
      begin
        Top := True;
        Bottom := True;
```

```
      end;
    end;

  if I.Mov_Type = NTSC then
    begin
      Left := True;
      Right := True;
      Top := True;
      Bottom := True;
      If (GOB mod 2) = 1 then
        begin
          if MAC in [1,1 + GOB_WID,1 + (GOB_WID * 2)] then
            begin
              Left := False;
            end
          else
            begin
              Left := True;
            end;
        end
      else
        begin
          if MAC in [GOB_WID,(GOB_WID * 2),(GOB_WID * 3)] then
            begin
              Right := False;
            end
          else
            begin
              Right := True;
            end;
        end;

      If GOB in [1,2] then
        begin
          if MAC in [1..GOB_WID] then
            begin
              Top := False;
            end
          else
            begin
              Top := True;
            end;
        end;

      If GOB in [9,10] then
        begin
          if MAC in [((GOB_WID*2)+1)..(GOB_WID*3)] then
            begin
              Bottom := False;
            end
          else
            begin
              Bottom := True;
            end;
        end;

      if not (GOB in [1,2,9,10]) then
        begin
```

```
          Top := True;
          Bottom := True;
        end;
    end;

if Top and (Pfe) then
  begin
    ConvToM_G(I.Mov_Type,(Idx1 - GOB_WID*2),GOB1,MAC1);
    MCA[2] := @((PPic[GOB1])^.Macs[MAC1]);
  end
else
  MCA[2] := nil;

if Top and Right and (Pfe) then
  begin
    ConvToM_G(I.Mov_Type,(Idx1 - GOB_WID*2 + 1),GOB1,MAC1);
    MCA[3] := @((PPic[GOB1])^.Macs[MAC1]);
  end
else
  MCA[3] := nil;

if Right and (Pfe) then
  begin
    ConvToM_G(I.Mov_Type,(Idx1 + 1),GOB1,MAC1);
    MCA[6] := @((PPic[GOB1])^.Macs[MAC1]);
  end
else
  MCA[6] := nil;

if Right and Bottom and (Pfe) then
  begin
    ConvToM_G(I.Mov_Type,(Idx1 + 1 + GOB_WID*2),GOB1,MAC1);
    MCA[9] := @((PPic[GOB1])^.Macs[MAC1]);
  end
else
  MCA[9] := nil;

if Bottom and (Pfe) then
  begin
    ConvToM_G(I.Mov_Type,(Idx1 + GOB_WID*2),GOB1,MAC1);
    MCA[8] := @((PPic[GOB1])^.Macs[MAC1]);
  end
else
  MCA[8] := nil;

if Bottom and Left and (Pfe) then
  begin
    ConvToM_G(I.Mov_Type,(Idx1 + GOB_WID*2 - 1),GOB1,MAC1);
    MCA[7] := @((PPic[GOB1])^.Macs[MAC1]);
  end
else
  MCA[7] := nil;

if Left and (Pfe) then
  begin
    ConvToM_G(I.Mov_Type,(Idx1 - 1),GOB1,MAC1);
    MCA[4] := @((PPic[GOB1])^.Macs[MAC1]);
  end
```

```
          else
            MCA[4] := nil;

          if Left and Top and (Pfe) then
            begin
              ConvToM_G(I.Mov_Type,(Idx1 - 1 - GOB_WID*2),GOB1,MAC1);
              MCA[1] := @((PPic[GOB1])^.Macs[MAC1]);
            end
          else
            MCA[1] := nil;

          if Pfe then
            MCA[5] := @((PPic[GOB])^.Macs[MAC]);

          Intra := DeMCSearch(MCA,UMac,CMac);

      end

end;

procedure SaveFrame(Var I : Infotype; var Pic : Rpictype; FNum : word);
{* This Procedure Callse Allocate and loads a Frame into Pic.          *}

var
  Idx1,Idx2,Idx3,Count,Blah1,Blah2 : word;
  MovType : byte;
  NumGOBS,NumMACS : word;
  MAC,MACRaw,MACRow,GOB,TrRow,CLine,CLine2 : word;
  InY : Rspicblktype;
  InU,InV : Rpicblktype;

  YFile : file of Rspicblktype;
  UFile : file of Rpicblktype;
  VFile : file of Rpicblktype;


begin
  MovType := I.Mov_Type;

  NumGOBS := Fr_Chars(MovType,'G');
  NumMACS := Fr_Chars(MovType,'M');

  assign(YFile,FFName(I.L_File_name,FNum,'Y'));
  assign(UFile,FFName(I.L_File_name,FNum,'U'));
  assign(VFile,FFName(I.L_File_name,FNum,'V'));

  rewrite(YFile);
  rewrite(UFile);
  rewrite(VFile);

  Count := 1;
  for Idx1 := 1 to NumMACS do
    begin
      for Idx2 := 1 to (BLK_WID*2) do
        begin

{* This Fiddely Bit Serves to copy a 16 x 16 Super block into the raw Frame *}
{* Because When its loaded the bytes are not arranged in blocks but in a    *}
```

Appendix B

```
{* Sequence, which gets placed in the load 16x16 block. Therefor this part *}
{* is required to convert this block in to the large frame size.          *}

          if MovType in [CIF,NTSC] then
            begin
              TrRow := ((Count - 1) div (GOB_WID * 2)) + 1;
              MACRaw := ((Count - 1) mod (GOB_WID * 2)) + 1;

              CLine := ((TrRow - 1) mod (BLK_WID * 2)) + 1;
              CLine2 := ((TrRow - 1) mod (BLK_WID)) + 1;
              MACRow := ((TrRow - 1) div (BLK_WID * 2)) + 1;

              if MACRaw > GOB_WID then
                begin
                  MACRaw := MACRaw - GOB_WID;
                  MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                  GOB := (((MacRow - 1) div GOB_HIG) + 1) * 2;
                end

              else
                begin
                  MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                  GOB := ((((MacRow - 1) div GOB_HIG) + 1) * 2) - 1;
                end;
            end
          else
            begin
              TrRow := ((Count - 1) div (GOB_WID )) + 1;
              MACRaw := ((Count - 1) mod (GOB_WID)) + 1;

              CLine := ((TrRow - 1) mod (BLK_WID * 2)) + 1;
              CLine2 := ((TrRow - 1) mod (BLK_WID)) + 1;
              MACRow := ((TrRow - 1) div (BLK_WID * 2)) + 1;

              MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
              GOB := (((MacRow - 1) div GOB_HIG) + 1);
            end;

          InY[Idx2] := Pic[GOB]^.Macs[MAC].RawMacYArray[CLine];

          Count := Count + 1;
        end;
      write(YFile,InY);                    {* Save Y Frame *}
    end;

{* This Fiddely Bit Serves to copy a 8 x 8 block into the raw Frame      *}
{* Because When its loaded the bytes are not arranged in blocks but in a  *}
{* Sequence, which gets placed in the load 8x8 block. Therefor this part  *}
{* is required to convert this block in to the large frame size.          *}


  Count := 1;
  for Idx1 := 1 to NumMACS do
    begin
      for Idx2 := 1 to (BLK_WID) do
        begin
          if MovType in [CIF,NTSC] then
            begin
```

```
            TrRow := ((Count - 1) div (GOB_WID * 2)) + 1;
            MACRaw := ((Count - 1) mod (GOB_WID * 2)) + 1;

            CLine := ((TrRow - 1) mod (BLK_WID)) + 1;
            CLine2 := ((TrRow - 1) mod (BLK_WID)) + 1;
            MACRow := ((TrRow - 1) div (BLK_WID)) + 1;

            if MACRaw > GOB_WID then
              begin
                MACRaw := MACRaw - GOB_WID;
                MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                GOB := (((MacRow - 1) div GOB_HIG) + 1) * 2;
              end

            else
              begin
                MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
                GOB := ((((MacRow - 1) div GOB_HIG) + 1) * 2) - 1;
              end;
          end
        else
          begin
            TrRow := ((Count - 1) div (GOB_WID )) + 1;
            MACRaw := ((Count - 1) mod (GOB_WID)) + 1;

            CLine := ((TrRow - 1) mod (BLK_WID)) + 1;
            CLine2 := ((TrRow - 1) mod (BLK_WID)) + 1;
            MACRow := ((TrRow - 1) div (BLK_WID)) + 1;

            MAC := ((MACRow - 1) mod GOB_HIG) * (GOB_WID) + MACRaw;
            GOB := (((MacRow - 1) div GOB_HIG) + 1);
          end;

        InU[Idx2] := Pic[GOB]^.Macs[MAC].RawMacUArray[CLine];
        InV[Idx2] := Pic[GOB]^.Macs[MAC].RawMacVArray[CLine];

        Count := Count + 1;
      end;
    write(UFile,InU);
    write(VFile,InV);
  end;
 close(YFile);
 close(UFile);
 close(VFile);
end;



procedure Decode(var I : Infotype);
{* This Procedure Does the huge Mega Decompression Process.          *}

var
  Idx1 : word;
  BS : Bstype;
  FSize : single;
  UFrame : Pictype;
  CFrame,PFrame : Rpictype;
  Chk : boolean;
```

```
  Pfe : Boolean;

begin
  FSize := FileExists(I.C_File_Name);
  assign (BS.BsFile,I.C_File_Name);
  reset (BS.BsFile);

  BS.FSize := FSize;
  BS.VBits := 0;
  BS.Tr_Bits := '';
  BS.CPos := 0;
  BS.FEOF := False;
  Idx1 := 0;
  Pfe := False;

  while (not BS.FEOF) or (BS.Vbits > 0) do
    begin
      Chk := ReadPicHeader(I,BS,Idx1);

      if Not Chk and not BS.FEOF then
        Die('Invalid Picture Header Detected',10);

      if not (I.Mov_type in [CIF,QCIF,NTSC]) then
        Die('Frame Type Invalid',12);

      if Not BS.FEOF then
        begin
          writeln;
          write('Allocating Work Space For Frame :',Idx1);
          AllocFrame(UFrame,I.Mov_Type);
          write(' ... Complete');

          writeln;
          write('Allocating current Load Frame :',Idx1);
          RAllocFrame(CFrame,I.Mov_Type);
          write(' ... Complete');

          writeln;
          write('Loading Frame :',Idx1,' ');
          BSFrameLoad(I,BS,UFrame);
          write(' ... Complete');

          writeln;
          write('De Quantising And IDCTing Frame :',Idx1,' ');
          DeQuant_IDCT(I,UFrame);
          write(' ... Complete');

          writeln;
          write('De_Motion Compensating Frame:',Idx1,' ');
          DeMotion(I,UFrame,CFrame,PFrame,Pfe);
          write(' ... Complete');

          writeln;
          write('Saving Frame :',Idx1);
          SaveFrame(I,CFrame,Idx1);
          write(' ... Complete');

          writeln;
```

```
          write('Un_Allocating Work Space For Frame :',Idx1);
          UnloadFrame(I,UFrame);
          write(' ... Complete');

          if not Pfe then
            begin
              PFrame := CFrame;
              Pfe := True;
            end
          else
            begin
              RUnloadFrame(I,PFrame);
              PFrame := CFrame;
            end;

          Idx1 := Idx1 + 1;
        end;
    end;
  close (BS.BsFile);
end;



begin {* Main Encode Program *}
  Init(Info);                    {* Init All Structures and Tables *}
  Decode(Info);                    {* Start the Encode Process     *}
  writeln;
  write('Program Terminated Gracefully, Press Enter To Exit ->');
  readln;
end.  {* Main Encode Program *}
```